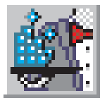
# Serving Up The Registry



*by David Baer*

Although working on a large, multi-year, development project (as I do) can be rewarding in many ways, there's a downside as well. There are a number of common practices which are often second nature to developers who produce small applications with some frequency. But when one rarely engages in such activities, it's not unusual to struggle a bit with tasks other developers could do almost in their sleep.

One such activity (at least for me) is programming registry access. Every time I've needed to do this, in the process of writing some small utility function that wasn't integrated into the main application, I've had to hunt down the last bit of code I wrote (usually six or more months earlier) to recall how it was done. It occurred to me that this area could use some automation. Of course, I could have done what many would do: fire up the browser and check out Torry's or the Delphi Super Page to see what other solutions might be available. But miss an opportunity to build a new component? Where's the fun in that?

No, it was time to engage in the agreeable pastime of component creation. What follows is a presentation of that component: what it does and how it was built. There will be little new information for seasoned component builders here, but you might just want to stick around for the first part. You may decide that `TRegValet` is a rather useful little tool to have in your toolkit.

## Amongst Our Weaponry

The initial idea was quite simple. I wanted something to allow the definition of a registry key, plus a set of value names and default value settings for them at design-time. I wanted these to be created the first time the program was executed and automatically stored at program termination. I wanted to have them loaded automatically during subsequent executions, and I wanted read/write access to the values to be almost as easy as accessing form properties. So far, so good.

By the way, I don't mean to dismiss Delphi's `TRegistry` class as inadequate. It's powerful and elegantly wraps some slightly ungainly API interfaces. But my needs are normally much simpler. Most small applications or utilities need only to store a few values (and always string type values) in one place, usually somewhere under `HKEY_CURRENT_USER\Software`. Furthermore, the registry access required by such programs is rarely in a performance-critical section of code. We don't need a super-optimized solution in most cases.

As always, having sat down at the drawing board, a few additional features began to suggest themselves. The first had to do with one very commonly used bit of information kept for the next time the program is launched: initial form placement and size. Given the frequency with which this sort of thing is useful, it would be nice to provide an especially easy-to-use service to handle it.

Another routine commission of the registry is storing file history lists. Here too, facilities tailored to serve this common requirement would be of benefit. This usage goes beyond typical operations (insert, find, etc) of a string list. We'll see shortly how this was addressed.

Of course, the first thing to do was resolve perhaps the most critical design decision a component builder faces. What class to descend from, maybe? No. I'm talking about what graphic to use for the component. After all, it's your work mounted for display on the component palette like some painting in the Tate Gallery. You want to draw potential users in, not have them skip over your brilliant creation because it's got some vapid image on it.

So, I thought of something like a reduced image of the highly recognizable Microsoft registry icon, placed on a platter being offered by some formally attired serving person. 'After all, how hard could that be?', I naively muttered (a sure-fire curse if there ever was one). Two hours after opening the image editor, I had to console myself with the thought that maybe I'm not much of an artist, but then, when was the last time David Hockney had to write a piece of software?

## Will That Be All, Sir?

Determining the best class to descend from was easy. This is a non-visual component, and the use of `TComponent` as the base class is just fine. Defining the necessary properties wasn't much more difficult. Naturally, we'll inherit the `Name` and (oft-abused) `Tag` properties. We need just three more. The first is a string property `Path`. I decided to hard code `HKCU` (ie `HKEY_CURRENT_USER`) as the major registry key, so `Path` specifies the location within `HKCU`. To give the component user some initial direction, I initialize the value of this property to `\Software\ ACompany\ AnApp`.

The next property is the main one: `Items`. This is a collection, the members of which define the registry value names, and either an optional default value or a maximum occurence count. The intent is to allow the component user to define a registry value as either a single value item or a list.

If a single value item is used, a default value may be specified. The first time the program is run, this value will be available to the client program, and it will be stored in the registry when execution terminates (perhaps as modified by the program during execution). Access to values in the application code is done via a public property, Values, which is accessed (for both read and write) using a parameter containing the item's name.

Alternatively, an item may be defined as a list by specifying a maximum occurrence count (more than 1). In this implementation, a list item may not be given a default value (I don't believe this would be that useful in most cases). List item values are accessed via the property IndexedValues, which takes a name and an index parameter. Another read-only property, Count, takes a name parameter and gives the number of items in the list.

TRegValet is fairly forgiving with respect to non-list items. If the client code requests a value for a non-existent item, it returns an empty string. If the client code assigns a value to a non-existent item, the component adds it (and stores it in the registry at the end of program execution).

List items, on the other hand, get a bit more scrutiny. New list items may not be defined on the fly. They must have been identified as such in the designer. Attempting to access a list item as a non-list, or vice versa, will trigger an exception.

Listing 1 shows the declaration of TRegValet along with the TCollectionItem and TCollection derivative classes it uses (all of the code for both the component and the demo program can be found on this issue's accompanying disk). For a quick scan, just pay attention to the public and published properties. TRegValet has one peculiar (but probably not significant) quality. The published properties are of use only at design-time. All the important runtime information is accessed via the public properties.

So far, I've described the properties of the component, but not any methods. Most of these (the public methods) supply support for list item manipulation. We'll discuss these in the context of the example application using TRegValet that follows.

## Proof In The Pudding

To demonstrate the component in action, I've spruced up a simple hex viewer program I had lying around. The program takes a file and displays it in a rich edit control, each line being formatted to include the hex offset, the hex display values and the printable values. I added options to allow the user to specify a display of 16 or 32 characters per line and to specify three relative font sizes. These settings will be retained between program executions in the registry, which of course is the point of this exercise. In addition, the client form is made to first appear in the same screen location and the same size as it was when last terminated.

Finally, I provided for the retention of a file history to be presented under the File menu item, offering the ability to reopen one of the last ten files opened in the viewer. Figure 1 shows the viewer

➤ *Listing 1*

```
ERegValetError = class(Exception);
TRegValetItem = class(TCollectionItem)
private
  FName: String;
  FDefaultValue: String;
  FMaxOccurs: Integer;
protected
  procedure SetDefaultValue(const Value: String);
  procedure SetName(const Name: String);
  procedure SetMaxOccurs(Value: Integer);
public
  constructor Create(Collection: TCollection); override;
  procedure Assign(Source: TPersistent); override;
published
  property DefaultValue: String read FDefaultValue
    write SetDefaultValue;
  property MaxOccurs: Integer read FMaxOccurs
    write SetMaxOccurs default 1;
  property Name: String read FName write SetName;
end;
TRegValetItems = class(TCollection)
private
  FOwner: TPersistent;
protected
  function GetItem(Index: Integer): TRegValetItem;
  procedure SetItem(Index: Integer; Item: TRegValetItem);
  function GetOwner: TPersistent; override;
public
  constructor Create(Owner: TPersistent);
  function Add: TRegValetItem;
public
  property Items[Index: Integer]: TRegValetItem
    read GetItem write SetItem; default;
end;
TRegValet = class(TComponent)
private
  FEasyRegItems: TRegValetItems;
  FListDelim: String;
  FPath: String;
  FSuppressSave: Boolean;
  Names: TStringList;
  NonIndexedValues: TStringList;
  function GetCount(const Name: String): Integer;
  function GetIndexedListRef(const Name: String):
    TStringList;
  function GetIndexedValue(const Name: String; Index:
    Integer): String;
  function GetValue(const Name: String): String;
  function ItemIsIndexed(const Name: String): Boolean;
  procedure SetIndexedValue(const Name: String;
    Index: Integer; const Value: String);
  procedure SetListDelim(const Value: String);
  procedure SetValue(const Name: String; const Value:
    String);
  procedure StoreIndexedItem(const Name: String; Value:
    String);
  procedure StoreItemFromReg(const Name: String; const
    Value: String);
  procedure StoreNonIndexedItem(const Name: String;
    const Value: String);
protected
  procedure Loaded; override;
  procedure MergeDefaults;
  procedure ReadRegistry;
  procedure WriteRegistry;
public
  constructor Create(AOwner: TComponent); override;
  destructor Destroy; override;
  procedure DeleteIndexedValue(
    const Name: String; Index: Integer); overload;
  procedure DeleteIndexedValue(const Name: String;
    const Value: String); overload;
  function IndexOf(const Name: String; const Value: String):
    Integer;
  procedure Initialize;
  procedure InsertIndexedValue(const Name: String;
    BeforeIndex: Integer; const Value: String);
  procedure MoveIndexedValueToFront(const Name: String;
    const Value: String); overload;
  procedure MoveIndexedValueToFront(const Name: String;
    Index: Integer); overload;
  procedure RestoreFormBounds;
  procedure SaveFormBounds;
public
  property Count[const Name: String]: Integer read GetCount;
  property SuppressSave: Boolean read FSuppressSave
    write FSuppressSave;
  property IndexedValues[const Name: String; Index:
    Integer]: String
    read GetIndexedValue write SetIndexedValue;
  property Values[const Name: String]: String read GetValue
    write SetValue; default;
published
  property Items: TRegValetItems read FEasyRegItems
    write FEasyRegItems;
  property Path: String read FPath write FPath;
  property ListDelim: String read FListDelim
    write SetListDelim;
end;
```

form with the File menu item opened.

The form's TRegValet component, named rvJeeves, has three items defined: one named FontSize has a default value of 10, one named CharsPerLine has a default value of 16, and one named FileHistory is a list item having a maximum occurrence value of 10. The initial form placement and sizing doesn't require an item, as we shall see shortly.

Listing 2 contains the relevant pieces of code from the viewer program, illustrating the component's use. Note that no code is needed to retrieve values from the registry at program startup or to return values to it at termination. The component performs both of these services automatically.

Let's begin with the form placement requirements. To accomplish this, we just need two method calls. The program may issue a call to RestoreFormBounds from FormShow, and a call to SaveFormBounds from FormClose. That's all there is to it.

The form's method InitializeFromRegistry also shows how the registry settings for font size and characters per line are obtained. The component's Values property is the default property, so rvJeeves[CHARS_PER_LINE] in the example code refers to that named item. Modifying values is just as straightforward. The form method FontSizeClick shows how a change to this value is communicated to the component.

Lastly, let's see the file history list in action. The form method SetFileMenuItems is executed when the File menu item is clicked. The code picks up the count of items in the history list, and initializes menu item Caption and Visible properties with information from the list.

Clicking one of the file names in this menu gets us into the LoadFile form method. Here we see that a failed attempt will cause the file to be removed from the history list:

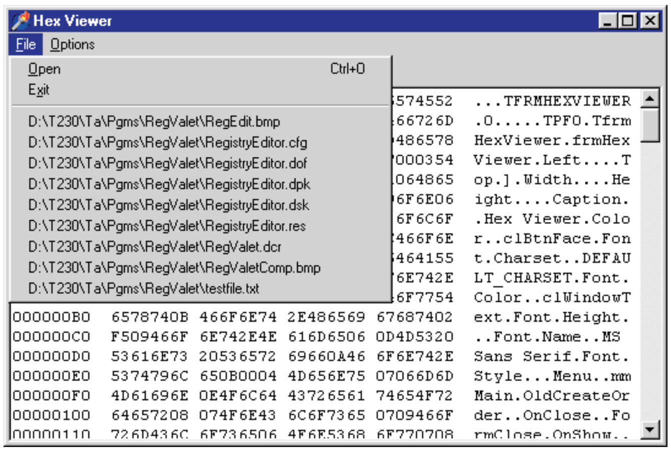
```
rvJeeves.DeleteIndexedValue(
  FILE_HISTORY, FileName)
```

A successful attempt will cause it to be moved to the front of the list:

```
rvJeeves.MoveIndexedValueToFront(
  FILE_HISTORY, CurrFile)
```

This move-to-front operation is pretty non-judgmental. If the name

➤ *Figure 1*


Hex Viewer
File
Options
Open Ctrl+O
Exit
D:\T230\Ta\Pgms\RegValet\RegEdit.bmp
D:\T230\Ta\Pgms\RegValet\RegistryEditor.cfg
D:\T230\Ta\Pgms\RegValet\RegistryEditor.dof
D:\T230\Ta\Pgms\RegValet\RegistryEditor.dpk
D:\T230\Ta\Pgms\RegValet\RegistryEditor.dsk
D:\T230\Ta\Pgms\RegValet\RegistryEditor.res
D:\T230\Ta\Pgms\RegValet\RegValet.dcr
D:\T230\Ta\Pgms\RegValet\RegValetComp.bmp
D:\T230\Ta\Pgms\RegValet\testfile.txt

```
                                            574552  ...TFRMHEXVIEWER
                                            66726D  .O.....TPFO.Tfrm
                                            486578  HexViewer.frmHex
                                            000354  Viewer.Left....T
                                            064865  op.].Width....He
                                            6F6E06  ight....Caption.
                                            6F6C6F  .Hex Viewer.Colo
                                            466F6E  r..clBtnFace.Fon
                                            464155  t.Charset..DEFAU
                                            6E742E  LT_CHARSET.Font.
                                            6F7754  Color..clWindowT
000000B0  6578740B 466F6E74 2E486569 67687402  ext.Font.Height.
000000C0  F509466F 6E742E4E 616D6506 0D4D5320  ..Font.Name..MS
000000D0  53616E73 20536572 69660A46 6F6E742E  Sans Serif.Font.
000000E0  5374796C 650B0004 4D656E75 07066D6D  Style...Menu..mm
000000F0  4D61696E 0E4F6C64 43726561 74654F72  Main.OldCreateOr
00000100  64657208 074F6E43 6C6F7365 0709466F  der..OnClose..Fo
00000110  726D436C 6F736506 4F6E5368 6F770708  rmClose.OnShow..
```

➤ *Listing 2*

```
const
  SMALL_FONT_SIZE = 8;
  NORMAL_FONT_SIZE = 10;
  LARGE_FONT_SIZE = 12;
  FILE_HISTORY = 'FileHistory';
  FONT_SIZE = 'FontSize';
  CHARS_PER_LINE = 'CharsPerLine';
procedure TfrmHexViewer.FormClose(Sender: TObject;
  var Action: TCloseAction);
begin
  rvJeeves.SaveFormBounds;
end;
procedure TfrmHexViewer.FormShow(Sender: TObject);
begin
  InitializeFromRegistry;
  SetMenuItemChecks;
  reData.Font.Size := FontSize;
end;
procedure TfrmHexViewer.InitializeFromRegistry;
var Size: String;
begin
  rvJeeves.RestoreFormBounds;
  CharsPerLine := 16;
  if rvJeeves[CHARS_PER_LINE] = '32' then
    CharsPerLine := 32;
  Size := rvJeeves[FONT_SIZE];
  if Size = '' then
    Size := IntToStr(NORMAL_FONT_SIZE);
  FontSize := StrToInt(Size);
end;
procedure TfrmHexViewer.FontSizeClick(Sender: TObject);
begin
  if Sender = miSmallFont then
    FontSize := SMALL_FONT_SIZE
  else if Sender = miNormalFont then
    FontSize := NORMAL_FONT_SIZE
  else if Sender = miLargeFont then
    FontSize := LARGE_FONT_SIZE;
  rvJeeves[FONT_SIZE] := IntToStr(FontSize);
  SetMenuItemChecks;
  reData.Font.Size := FontSize;
end;
procedure TfrmHexViewer.SetFileMenuItems(Sender: TObject);
var
  Cnt: Integer;
  procedure SetMenuItem(Index: Integer; Item: TMenuItem);
  begin
    Item.Visible := (Index < Cnt);
    if Index < Cnt then begin
      Item.Caption :=
        rvJeeves.IndexedValues[FILE_HISTORY, Index];
      if Item.Caption = CurrFile then
        Item.Visible := False;
    end;
  end;
begin
  Cnt := rvJeeves.Count[FILE_HISTORY];
  SetMenuItem(0, miFile0);
  SetMenuItem(1, miFile1);
  SetMenuItem(2, miFile2);
  SetMenuItem(3, miFile3);
  SetMenuItem(4, miFile4);
  SetMenuItem(5, miFile5);
  SetMenuItem(6, miFile6);
  SetMenuItem(7, miFile7);
  SetMenuItem(8, miFile8);
  SetMenuItem(9, miFile9);
  miSep1.Visible := (miFile0.Visible or miFile1.Visible);
end;
procedure TfrmHexViewer.LoadFile(const FileName: String);
var MS: TMemoryStream;
begin
  if FileName <> '' then begin
    MS := TMemoryStream.Create;
    try
      try
        MS.LoadFromFile(FileName);
      except
        rvJeeves.DeleteIndexedValue(FILE_HISTORY, FileName);
        raise;
      end;
      FormatFile(MS, CharsPerLine, reData.Lines);
      CurrFile := FileName;
      lblFileName.Caption := CurrFile;
      rvJeeves.MoveIndexedValueToFront(
        FILE_HISTORY, CurrFile);
    finally
      MS.Free;
    end;
  end;
end;
```

is in the list, it is moved to the front. If not, it is added at the front, jettisoning the last entry if the list is at its maximum capacity. Several other routines, including `InsertIndexedValue` and the overloaded `DeleteIndexedValue`, are available for more complicated list item manipulation.

There are two additional things left to discuss. The first has to do with how list values are maintained in the registry. These are concatenated into a string value, with the various entries delimited by a predefined character string, which, by default, is a single semicolon (;). For lists of file names, this should work just fine. However, if this delimiter is inappropriate for the data content, an alternative delimiter may be specified in the published component property `ListDelim`.

Finally, if for some reason the application needs to suppress saving of the managed values back to the registry at program termination, the public property `SuppressSave` may be set to `True` for this purpose.

➤ *Listing 3*

```
procedure TRegValet.Loaded;
begin
  inherited Loaded;
  Initialize;
  MergeDefaults;
  ReadRegistry;
end;
procedure TRegValet.Initialize;
var
  I: Integer;
  ERI: TRegValetItem;
  L: TStringList;
begin
  for I := 0 to (FEasyRegItems.Count - 1) do begin
    ERI := FEasyRegItems[I];
    Names.Add(ERI.Name);
    NonIndexedValues.Add(ERI.DefaultValue);
    if ERI.MaxOccurs > 1 then begin
      L := TStringList.Create;
      L.Capacity := ERI.MaxOccurs;
      Names.Objects[I] := L;
    end;
  end;
end;
procedure TRegValet.MergeDefaults;
var
  I: Integer;
  ERI: TRegValetItem;
begin
  for I := 0 to (FEasyRegItems.Count - 1) do begin
    ERI := FEasyRegItems[I];
    if ERI.DefaultValue <> '' then
      Values[ERI.Name] := ERI.DefaultValue;
  end;
end;
procedure TRegValet.ReadRegistry;
var
  I: Integer;
  Reg: TRegistry;
  Names: TStringList;
begin
  Reg := TRegistry.Create;
  try
    Reg.RootKey := HKEY_CURRENT_USER;
    if not Reg.OpenKeyReadOnly(FPath) then
      Exit;
    Names := TStringList.Create;
    try
      Reg.GetValueNames(Names);
      for I := 0 to (Names.Count - 1) do
        StoreItemFromReg(
          Names[I], Reg.ReadString(Names[I]));
    finally
      Names.Free;
    end;
    Reg.CloseKey;
  finally
    Reg.Free;
  end;
end;
destructor TRegValet.Destroy;
var
  I: Integer;
begin
  if not FSuppressSave then
    WriteRegistry;
  FEasyRegItems.Free;
  for I := 0 to (Names.Count - 1) do
    if Names.Objects[I] <> nil then
      TStringlist(Names.Objects[I]).Free;
  Names.Free;
  NonIndexedValues.Free;
  inherited Destroy;
end;
procedure TRegValet.WriteRegistry;
var
  I: Integer;
  Reg: TRegistry;
  procedure WriteIndexedItems(const Name: String; L:
    TStringList);
  var
    I: Integer;
    S: String;
  begin
    for I := 0 to (L.Count - 1) do
      S := S + L[I] + FListDelim;
    Reg.WriteString(Name, S);
  end;
begin
  Reg := TRegistry.Create;
  try
    Reg.RootKey := HKEY_CURRENT_USER;
    if Reg.OpenKey(FPath, False) then
      Reg.DeleteKey(FPath);
    Reg.OpenKey(FPath, True);
    for I := 0 to (Names.Count - 1) do begin
      if Names.Objects[I] <> nil then
        WriteIndexedItems(
          Names[I], TStringList(Names.Objects[I]))
      else
        Reg.WriteString(Names[I], NonIndexedValues[I]);
    end;
    Reg.CloseKey;
  finally
    Reg.Free;
  end;
end;
procedure TRegValet.RestoreFormBounds;
var
  Left: Integer;
  Top: Integer;
  Height: Integer;
  Width: Integer;
  BoundsStr: String;
  function NextValue: Integer;
  var I: Integer;
  begin
    Result := -9999;
    I := Pos(';', BoundsStr);
    if I > 0 then begin
      Result := StrToInt(Copy(BoundsStr, 1, (I - 1)));
      BoundsStr := Copy(BoundsStr, (I + 1), $7FFF);
    end else
      // values been manually altered, give up
      BoundsStr := '';
  end;
begin
  if (Owner is TCustomForm) then begin
    BoundsStr := Values['RegValetFormBounds'];
    if BoundsStr <> '' then begin
      try
        Left := NextValue;
        Top := NextValue;
        Width := NextValue;
        Height:= NextValue;
        if (Left <> -9999) and (Top <> -9999) and
          (Width <> -9999) and (Height <> -9999) then
          TCustomForm(Owner).SetBounds(
            Left, Top, Width, Height);
      except
        // don't make a big deal out of it
      end;
    end;
  end;
end;
procedure TRegValet.SaveFormBounds;
var
  Placement: TWindowPlacement;
  R: TRect;
begin
  if (Owner is TCustomForm) then begin
    Placement.Length := SizeOf(TWindowPlacement);
    GetWindowPlacement(
      TCustomForm(Owner).Handle,@Placement);
    R := Placement.rcNormalPosition;
    Values['RegValetFormBounds'] := IntToStr(R.Left) + ';' +
      IntToStr(R.Top) + ';' +
      IntToStr(R.Right - R.Left) + ';' +
      IntToStr(R.Bottom - R.Top) + ';';
  end;
end;
```

## Wrap It Up For You, Sir?

In the space that remains, I'll spend a little time describing the internals of TRegValet, or at least those aspects that are off the beaten track. As you might expect, we'll make use of internal Delphi TRegistry objects to do the actual registry access dirty work.

To begin with, let's focus on the automated registry retrieval and storage services. Although I don't take any steps in the code to prevent this, TRegValet is useful only when added to an application at design-time. Creating an instance dynamically doesn't make sense for this kind of component.

Therefore, it's safe to assume that the Loaded method will be called, and it's here we place the code to do the initial retrieval. Listing 3 contains some of the component code being discussed here. We can also let the registry writing take place immediately before destruction in an overridden Destroy.

The value information is stored in one or more internal TStringList objects. The main list, Names, stores all the item names for both non-list and list items. A parallel list, NonIndexedValues, retains the associated values for the non-list items. The Data property of Names is used for list items, storing a reference to a TStringList containing the individual values of items in the list. There is one of these string lists for each list item type.

The form placement services can be seen in the methods RestoreFormBounds and SaveFormBounds. Placement values are stored as four concatenated integers in a value named RegValetFormBounds. In saving these values, we do not want to use the owner form's Top, Left, Width and Height properties. If the form is maximized, these will contain the actual values of the maximized state. If the client form is terminated in this state, it will have these values when next launched (that is it will appear as maximized), which is probably not what we want. Even worse, the maximize/restore button will display the maximize graphic. Instead, when saving the placement data, the component calls the Windows API routine GetWindowPlacement to acquire true normal placement values.

Most of the remaining code in TRegValet manages the access to the values. A lot of string list access takes place, but none of it should be too mysterious to anyone familiar with these things.

Before wrapping up, I'll offer a few comments about the use of the TCollection and TCollectionItem derivatives used in the component. Based on the occasional question I see in the Borland component writing newsgroup on this subject, their proper use for supporting published aggregate properties is not all that obvious to those first trying to work with them. Although there are good examples to follow in the VCL code, the Delphi help is not at its best in this area.

The collection used for Items in TRegValet was my first use of this device since Delphi 2, and I was pleased with how nicely Borland improved them by solving the problem of supplying a generic property editor. It wasn't too difficult to examine several VCL implementations to get the TRegValetItems collection to work and to successfully summon up the default collection property editor.

Here's a simple recipe for writing a minimal collection and collection item class that can call upon the default Delphi collection item editor. You do need to declare both a TCollectionItem and a TCollection derivative class. Assuming your collection items have only properties of the form MyProp: ... read FMyProp write FMyProp (that is, no property read or write methods), you need no overridden methods for the TCollectionItem derivative class. For completeness, however, you should supply an Assign method modelled on the code in TRegValetItems. The TRegValetItem class does supply other methods, but they aren't required for a minimal working implementation.

For the TCollection derivative class, you will need the Create, GetItem, GetOwner and SetOwner methods. Note in the code for TRegValetItems those which are declared override and those which are not. The first three of these methods can be used almost verbatim, with only the class types changed as appropriate. TRegValetItems.SetItem requires only the inherited SetItem call for a minimal working implementation (the remainder of the code in that method is specific to this implementation). Finally, you should supply an Add method for completeness. Do these things, and you should have a working TCollection of your own.

## Valediction

Lessons learned? Well, we all know it's advisable to cultivate a personal friendship with an attorney or a general building contractor whenever the opportunity presents itself. To that list, I think component developers ought to add the profession of graphic artist.

In the *'Duh!'* category, I discovered that it's a *very* good idea when debugging code that modifies the registry to *not* have the registry editor open at the same time to monitor your progress.

More seriously, I believe this exercise illustrates that if you often use a complex class or component and find yourself routinely using only the same 20% of its capabilities, it may be an excellent candidate for wrapping within a service layer to make your life easier.

Hmmm... TImageList gives me fits every time I try to use it, and I'm never trying to do anything particularly fancy. Maybe that's a good one to build a wrapper layer around. After all, how hard could that be?

David Baer is Chief Software Architect at Spear Technologies in San Francisco. He's never known a real valet or butler, but he has seen all of the episodes of *Upstairs, Downstairs* at least twice. He can be reached at dbaer@speartechnologies.com