Diagramming In Delphi

by Jim Cooper

A picture is worth a thousand words, they say. I tried to think of a few I could draw, instead of writing this article, but as you can see, my artistic abilities weren't up to the task. While it may be some time before I have anything hanging in a major gallery, there is no doubt that some programs can benefit greatly from diagrams. This article is intended to present a set of classes that provide basic diagramming capabilities, and an example of their use. For our purposes, a diagram will be defined as a set of nodes, optionally joined by some sort of connector. Something like a network diagram, a class hierarchy, or a finite state machine diagram.

There will be two main sections. Firstly, the description of the diagramming classes. Initially, these will only provide static diagramming abilities, that is, the elements of a diagram cannot be moved or resized by a user, only by code. Editor willing, the next article will show how they can be extended to provide movable, resizable, streamable diagramming elements. Secondly, an example program. A small application will be developed to give a visual map of a website.

Picture This

No, we're not going through a list of Blondie songs, *à la* Julian Bucknall. I just put it in to panic the Editor. What we're really going to do is describe the base diagramming class. It's a descendant of `TGraphicControl`. The reasons for this are several. Primarily, it provides a `TCanvas` to draw on, but doesn't have a Windows handle and therefore doesn't use so many Windows resources. This is not such an issue as it once was, but it's good practice to minimise resource use whenever possible. Even though they cannot receive the focus, `TGraphicControls` still receive mouse messages, which will be useful even in a 'static' diagram. Just you wait and see. Secondly, several useful properties and methods are already provided. Thirdly, the diagramming objects will automatically be able to be streamed, simply by making the relevant properties published. Lastly, Windows knows when to draw these controls, so we only need to override the `Paint` method, not worry about when to draw the control.

Creating our own base class is also a good idea, because the basis of all the diagramming components functionality can be provided in the one place. For instance, to make creation of these diagramming objects easier, the base class will provide a unique naming scheme. Similar examples are scattered throughout the VCL, where nearly all components are descended from a `TCustomXXXX` class. Although the diagramming objects we will build are components, they are not registered to appear on the component palette. It's not that they can't be, just that I don't see the point. If you want to, go right ahead. My intention is that they will normally be used programmatically, either being created as required (as in the web mapper application discussed later), or by the user in some sort of diagramming program. In this latter case, more functionality will probably be required, especially moving and resizing diagram elements, and storing the diagrams.

Note also that we will use the class naming convention suggested way back when in this magazine. A short prefix between the `T` and the descriptive part of the class name is used to increase the chances of the name being unique. I also tend to use the prefix at the start of the unit names my classes are in. In this case, we will use `jim`. I recommend this practice, as I know of three `TdbTreeView` controls, for instance. You can register these prefixes for free at the Delphi Prefix Registry, run by Steven Healey at:

```
Http://developers.href.com/
  registry/dpr.htm
```

This is an excellent service, intended to ensure that nobody uses the same prefixes.

The main inherited properties that we will be making use of are `Canvas`, `Owner` and `Parent`. Note that the `Owner` of a component is the form on which it is placed. This is different from a `Parent`, which may be a container control, like a `TPanel` or `TScrollbox`. Setting the `Parent` is necessary for the control to appear on a form.

We will need to override some virtual methods. The most important are `Create`, which is used to implement a unique naming scheme and to allocate any resources needed, and `Destroy`, which releases those resources. We also need to overide `Paint`, to provide the particular drawing services required. Another useful method to override is `SetBounds`, which sets all the component's boundary properties at one time,

➤ *Listing 1*

```
TjimCustomShape = class(TGraphicControl)
  // All controls descend from this, to help with streaming and unique naming
  protected
    procedure SetBounds(ALeft,ATop,AWidth,AHeight : Integer); override;
  public
    constructor Create(AOwner : TComponent); override;
  published
    // Make these properties available
    property OnClick;
    property OnDblClick;
  end;
```

```
interface
type
  TjimTextShape = class(TjimCustomShape)
  private
    FText     : string;
    FAutosize : Boolean;
    procedure SetText(Value : string);
    procedure SetAutosize(Value : Boolean);
  protected
    procedure Paint; override;
    procedure SetBounds(ALeft, ATop, AWidth, AHeight :
      Integer); override;
  public
    constructor Create(AOwner : TComponent); override;
  published
    property Text     : string read FText write SetText;
    property Autosize : Boolean
      read FAutosize write SetAutosize;
  end;
implementation
procedure TjimTextShape.Paint;
var TempRect : TRect;
begin {Paint}
  inherited Paint;
  if not Assigned(Parent) then begin
    Exit;
  end;
  TempRect := ClientRect;  // So can pass as a var parameter
  DrawText(Canvas.Handle,PChar(FText),Length(FText),
    TempRect, DT_CENTER or DT_NOPREFIX or DT_WORDBREAK);
end;  {Paint}
procedure TjimTextShape.SetBounds(
  ALeft,ATop,AWidth,AHeight: Integer);
begin {SetBounds}
  // Check that the control bounds are sensible. Note that
  // this also works if try to set Left, Top etc properties,
  // as their access methods call
  // SetBounds().
  if FAutosize and Assigned(Parent) then begin
    NoLessThan(AWidth,Canvas.TextWidth(FText));
    NoLessThan(AHeight,Canvas.TextHeight(FText));
  end;
  inherited SetBounds(ALeft,ATop,AWidth,AHeight);
end;  {SetBounds}
```

➤ *Listing 2*

ensuring that repainting will not occur between setting each of them. This method is also called when `Left`, `Width`, `Top` and `Height` are set, so it is a good place to enforce any maximum or minimum sizes, among other things.

Let's look at the code for the declaration of the base class (shown in Listing 1).

This is very simple. The constructor is overridden so that the unique naming scheme can be implemented. Because these are components, they must be named uniquely if they are to be placed on a form (and for streaming). In essence, all we will do is create a component name (the word `Shape` followed by an integer), check whether it is already in use and, if it is, to continue testing names until we find one that is unused. We will not rely on the VCL raising an exception if we use a duplicate name, because the class of exception used can actually be raised in other circumstances that we might want to detect. Note that the ability to stream a diagram object is now built in because they will all be descended from `TComponent`, which has that functionality. Groovy, this object orientation. We can store diagrams in a file with no work. Actually, there is a small amount to do, but we'll get to that next time. The astute amongst you will have noticed that `SetBounds` is overridden as well. Fear not, we'll come back to it later. First, let's start to make some useful components.

1.001 Pictures

A text caption is a common diagramming element, but then I come from peasant stock myself. In the website mapper application, they will be used to display the various addresses. Captions will be instances of the `TjimTextShape` class defined in Listing 2.

The interesting features of this class are the overridden `Paint` and `SetBounds` methods. Their implementations are shown in Listing 2 as well. As expected, the `Paint` method draws the text held in the `Text` property. It uses the Windows API routine `DrawText` because you could play with the flags that constitute the last parameter if, say, you wanted to add different text justifications, or to turn word wrapping on and off. `SetBounds` is used to ensure the control grows and shrinks with the text, if the `Autosize` property is `True`. The `NoLessThan` procedure used in `SetBounds` is one of the ancillary routines used in the file JimShape.pas, which can be found on the disk. There are also some small routines for getting the height and width of rectangles, and the minimum and maximum of an array of integers. Note that both methods check that the component has a `Parent` (that is, it has been placed on a form), because otherwise the `Canvas` property will be invalid.

A Plethora Of Pictures

We will definitely want pretty bits on our diagrams, so we need a class to display bitmaps. `TjimBitmapShape` is defined in Listing 3.

We will go the same way as many of the VCL controls, and display an image held in a `TImageList`. There is no particular need for this, of course, and you can easily modify the class to use a `TPicture` instead, so you could store any bitmap, metafile or icon. One advantage of using an `ImageList` is that the images will be compiled into the application. This could also be done with a Windows resource file,

➤ *Listing 3*

```
TjimBitmapShape = class(TjimCustomShape)
  private
    FImages     : TImageList;
    FImageIndex : Integer;
    FCaption    : TjimTextShape;
    procedure SetImages(Value : TImageList);
    procedure SetImageIndex(Value : Integer);
    procedure SetCaption(Value : TjimTextShape);
  protected
    procedure SetBounds(ALeft,ATop,AWidth,AHeight : Integer); override;
    procedure Paint; override;
    procedure Notification(AComponent : TComponent;Operation : TOperation);
      override;
  public
    constructor Create(AOwner : TComponent); override;
  published
    property Images     : TImageList read FImages write SetImages;
    property ImageIndex : Integer read FImageIndex write SetImageIndex;
    property Caption    : TjimTextShape read FCaption write SetCaption;
  end;
```

but this way we do less work. Usually we will be using just a few images many times. The other is that it makes painting the control very easy with the methods of TImageList. Because there is a reference to another control, there must also be a Notification procedure, in case, for example, the ImageList is deleted.

The access methods are quite usual, making sure that values for FImages and FImageIndex are valid. We can use a few tricks, like setting FImages only if it is a new value, when it is used to set the size of the component to the image size. However, the main point of interest here is the overridden Paint method. Note that unless the component has been placed on a form, we should not attempt any drawing, similarly if the ImageList or image index are invalid.

A small point which you will notice if you examine the source on the disk is that the if statement that checks this assumes that 'lazy' evaluation of Boolean expressions is enabled, hence the {$B-} at the top of the file. Lazy evaluation stops as soon as the result is known. In this case, as soon as one of the subexpressions is true. This saves many nested if statements. This mode is on by default, but I prefer to ensure that it stays that way, so I explicitly turn it on in those files that rely on it. The last two lines of the method do the drawing, using a transparent drawing style. This is my preference, but isn't essential. For more tricks with ImageLists that might be useful, see David Collie's article in Issue 36.

The picture will often need a caption, so there is a reference to a TjimTextShape. The SetBounds method ensures that the caption is moved whenever the image is moved or resized. We will use a simple technique of aligning the left edge of the caption with the left edge of the image, and keeping it 5 pixels below. If you want a different alignment, this is the place to implement it.

I won't be describing any other types of node elements this time, but hopefully it's now fairly clear how to go about it. First, derive a new type from TjimCustomShape, or one of its descendants. Second, override the constructor and destructor if resources need to be allocated. Third, override the Paint method to do the sort of drawing you require. There are TCanvas methods to draw various shapes and colours, for instance. Fourth, override SetBounds if necessary. This might be to restrict maximum or minimum sizes, or to ensure that associated diagram components also get moved or sized. Lastly, override the Notification method if you reference any other controls.

Pointy Bits

We said at the beginning that a diagram consists of nodes and connectors. It now remains to develop the connector class. Because there are many different sorts of connectors, there is a need for another base class. To keep the discussion simple, we will consider a connector to be between two nodes only. So there will be some sort of line joining the nodes, possibly with some sort of terminator symbol at each end, an arrowhead, for instance.

The point of connection between a node and a connector is a little more complicated than it might appear at first. Obviously, it will be a point on a TjimCustomShape. To make diagrams look neater, we also want to be able to specify which side to

➤ *Listing 4*

```
interface
type
  TjimConnectionSide = (csLeft,csRight,csTop,csBottom);
  TjimConnection = class(TPersistent)
  private
    FShape  : TjimCustomShape;
    FSide   : TjimConnectionSide;  // Side to connect to
    FOffset : Integer; // Distance from top or left of side
  public
    constructor Create;
    procedure Assign(Source : TPersistent); override;
    // Gets connection point in parent's coordinates
    function ConnPoint(TerminatorRect : TRect): TPoint;
    // Gets terminator connection point in parent's
    // coordinates
    function TermPoint(TerminatorRect : TRect): TPoint;
    // Functions to get boundaries of the terminators
    function LeftMost(TerminatorRect : TRect): TPoint;
    function RightMost(TerminatorRect : TRect): TPoint;
    function TopMost(TerminatorRect : TRect): TPoint;
    function BottomMost(TerminatorRect : TRect): TPoint;
  published
    property Shape : TjimCustomShape
     read FShape write FShape;
    property Side   : TjimConnectionSide
     read FSide write FSide;
    property Offset : Integer read FOffset write FOffset;
  end;
  TjimConnector = class(TjimCustomShape)
  private
    FLineWidth  : Integer;
    // The shapes connected by this control
    FStartConn : TjimConnection;
    FEndConn   : TjimConnection;
    // Area of the terminator symbol to be drawn (in
    // horizontal position)
    FStartTermRect : TRect;
    FEndTermRect   : TRect;
    procedure SetLineWidth(Value : Integer);
    function  GetConn(Index : Integer) : TjimConnection;
    procedure SetConn(Index : Integer;
      Value : TjimConnection);
    function  GetTermRect(Index : Integer) : TRect;
    procedure SetTermRect(Index : Integer;Value : TRect);
  protected
    procedure Paint; override;
    procedure Notification(AComponent :
      TComponent; Operation : TOperation); override;
    // For drawing arrows etc. Called from Paint.
    procedure DrawStartTerminator; virtual;
    procedure DrawEndTerminator; virtual;
    // Restrict the minimum size
    procedure SetBounds(ALeft, ATop, AWidth, AHeight :
      Integer); override;
    // Converts point from parent's coordinates to own
    // coordinates
    function  Convert(APoint : TPoint) : TPoint;
    function  IsConnected(ConnectedShape :
      TjimCustomShape) : Boolean;
  public
    constructor Create(AOwner : TComponent); override;
    destructor  Destroy; override;
    procedure SetConnections(TheStartConn,TheEndConn :
     TjimConnection);
    // Called when moving one of the connected shapes
    procedure SetBoundingRect;
    property StartTermRect : TRect
      index 1 read GetTermRect write SetTermRect;
    property EndTermRect : TRect
      index 2 read GetTermRect write SetTermRect;
  published
    // Publish these properties so that component streaming
    // can be used to store them in a file
    property LineWidth : Integer read FLineWidth
      write SetLineWidth default 1;
    property StartConn : TjimConnection
      index 1 read GetConn write SetConn;
    property EndConn : TjimConnection
      index 2 read GetConn write SetConn;
  end;
```

connect to (top, bottom, left or right), and how far the connection point is offset from the top or left. The definitions of all the required classes are in Listing 4.

Notice that TjimConnection is descended from TPersistent. This is because we want to be able to stream the published properties, but we don't need any of the overheads of a component. The two public functions are used to help draw the terminator symbol for the connection, as they return the points where the connecting line joins the terminator symbol, and the terminator symbol joins the node component, respectively. TjimConnector uses two of these connection objects: StartConn and EndConn. The public properties StartTermRect and EndTermRect are used to define the size of the terminator symbols at each end of the connecting line. The protected virtual methods DrawStartTerminator and DrawEndTerminator draw the actual symbols. In this class, they are empty methods. They could be abstract methods, but that would make it impossible to create an instance of class TjimConnector, and you may want to just draw a connecting line, with nothing else at the ends. Descendant classes should override one or both of these methods.

Otherwise, for all the apparent complexity of the declaration, creating TjimConnector only follows the same procedure for deriving a new diagramming component that we outlined above. We derive a new type from TjimCustomShape, and the constructor and destructor are overridden to create and free the TjimConnection objects. The Paint method is overridden to do the required drawing. The connector is drawn only if both start and end shapes have been assigned, and the line is a straight line drawn between points calculated by the TjimConnection.ConnPoint method. The virtual terminator drawing methods are called as well. Then we override SetBounds to ensure that the control is at least as large as the line width, and at least as large as the largest of the terminating shapes.

```
interface
type
  TjimSingleHeadArrow = class(TjimConnector)
  protected
    procedure DrawArrowHead(ConnPt,TermPt : TPoint);
    procedure DrawEndTerminator; override;
  public
    constructor Create(AOwner : TComponent); override;
  end;
implementation
constructor TjimSingleHeadArrow.Create(AOwner : TComponent);
begin {Create}
  inherited Create(AOwner);
  EndTermRect := Rect(0,0,25,10);
end;  {Create}
...
procedure TjimSingleHeadArrow.DrawEndTerminator;
var
  ConnPt,TermPt : TPoint;
begin {DrawEndTerminator}
  inherited DrawEndTerminator;
  if Assigned(FEndConn.Shape) then begin
    ConnPt := Convert(FEndConn.ConnPoint(EndTermRect));
    TermPt := Convert(FEndConn.TermPoint(EndTermRect));;
    DrawArrowHead(ConnPt,TermPt);
  end;
end;  {DrawEndTerminator}
```

➤ *Listing 5*

```
procedure TjimCustomShape.SetBounds(ALeft,ATop,AWidth,AHeight : Integer);
var i : Integer;
begin {SetBounds}
  inherited SetBounds(ALeft,ATop,AWidth,AHeight);
  // Search for any connectors between this and any other control
  // First check that this control has been placed on a form
  if not Assigned(Parent) then begin
    Exit;
  end;
  // Search parent control for TjimConnector components
  for i := 0 to Parent.ControlCount - 1 do begin
    if Parent.Controls[i] is TjimConnector then begin
      with TjimConnector(Parent.Controls[i]) do begin
        // Check if this component is at either end of the connector
        if IsConnected(Self) then begin
          // Resize the connector
          TjimConnector(Parent.Controls[i]).SetBoundingRect;
        end;
      end;
    end;
  end;
end;  {SetBounds}
```

➤ *Listing 6*

This avoids any strange errors when doing the painting. Finally, we override the Notification method to check for deletion of either of the connected shapes.

We follow exactly the same procedure to create a connector with an arrowhead at the end of the line. This time, we inherit from TjimConnector, so the Paint, SetBounds and Notification procedures are already sufficient for our needs. We must override DrawEndTerminator to draw the arrowhead, as shown in Listing 5. Note that the DrawArrowHead method is made protected, so that descendant classes can call it. You could inherit a double headed arrow class from this one, for instance. The arrowhead will point in the required direction, depending on the values of ConnPt (the blunt end), and TermPt (the pointy end). The code for DrawArrowHead is straightforward but tedious, so we won't bother with it here. The other important point is that EndTermRect is set in the constructor. This ensures that the connecting line is drawn to the middle of the blunt end of the arrowhead. Any classes that are descended from TjimConnector must set StartTermRect and EndTermRect in their constructor, if a start or end terminator symbol is to going to be drawn, respectively.

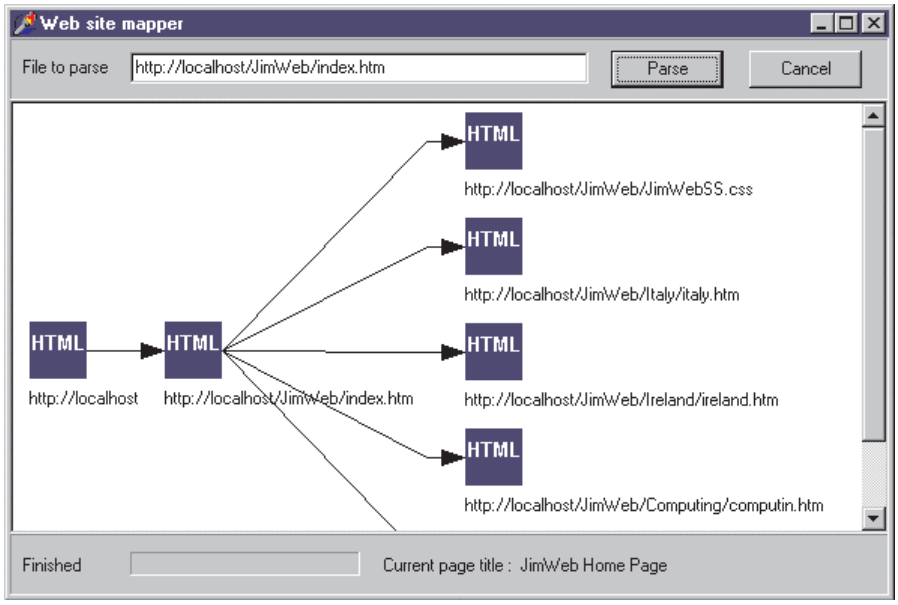
Everywhere That Mary Went

Remember that I said we would come back to the SetBounds method of TjimCustomShape? Well, now's the time. We have defined the connectors between node elements, but what happens when

one of those nodes moves? It would be nice if any connectors that are attached to it would automatically redraw themselves. This is exactly what happens in Listing 6.

First of all, the inherited `SetBounds` method (from `TGraphicControl`) is called to ensure the component actually gets resized or moved. Recall that `SetBounds` is called whenever `Left`, `Top`, `Width` or `Height` is set. If no `Parent` has been assigned, then this component has not yet been used in a diagram. If there is a `Parent`, then its `Controls` list is searched for `TjimConnector` components. Note that the `Components` list is not used, as it contains those components owned by a control, which is a bit different. Normally, all components on a form are owned by the form, whereas we just want those components that are on our drawing surface (usually a panel or scroll box). Anyway, if any `TjimConnectors` are found, they are checked to see if they are connected to the current component. If so, the connector control is resized and redrawn by calling its `SetBoundsRect` method. Because all node elements will be descended from `TjimCustomShape`, and all connectors will inherit from `TjimConnector`, and you will always call the inherited `SetBounds` method when you override it (you will, won't you?), you never need to worry about redrawing connectors. Just move the node elements and the connectors will follow all by themselves. Too easy, this inheritance stuff.

There are a couple of additional points that should be made clear. Firstly, any properties that would need to be stored, if we were providing such a facility, need to be made published. Next time, we will use that to store and retrieve diagrams. You should bear that in mind for any classes you build. Secondly, we have only included diagramming components that are basically rectangular. It is possible to have different shaped windows, and the techniques in the article by Steven Colagiovanni in Issue 25 could be adapted for use here.



➤ *Figure 1*

Here Be Dragons

We are now in a position to draw diagrams. To demonstrate how, we are going to develop a simple application for navigating through a website visually. Figure 1 shows it in action, looking at the local web pages on my machine. What we have are three categories of web page. The page second from the left (http://localhost/JimWeb/index.htm) is the current page. To its left is the page that contains the link we followed to get here, and to its right are all the links on the current page. From this view, you can plainly see that I'm a completely up-to-date web guru, because I'm using a style sheet (JimWebSS.css). If I were artistically gifted, you would also see that different icons are used to represent different URLs, like images, mailto or FTP addresses. (And yes, clever you, I have used the internet to research my holidays in Ireland and Italy.) Double clicking on a web page icon will make that the current page. The previous page is included so that it is possible to navigate back through the site. Because this is supposed to be a simple example, no sophisticated placement algorithm is used. The parent and current pages always appear in the middle of the scroll box, and the child pages are just added one below the other, because the scroll box will resize itself automatically to let them fit. Obviously, this scheme is too simplistic for pages with huge numbers of links, but you can easily change this to something more sophisticated if you wish.

The application also needs to connect to a server to get the pages. Microsoft's Personal Web Server is free, surprisingly small and easy to set up, and is what I use for the web pages on my local machine (it will also work as a web server on a small network). You can also use your internet connection to connect to external sites. In Issue 27, Dr. Bob did something similar in his broken links detector. You may wish to use his techniques for getting web pages, instead of the `TNMHttp` component we will be using. However, the parsing scheme he used was fairly simple, and I for one had problems with pages generated by FrontPage, which does truly bizarre things to the HTML formatting on occasion.

I had originally intended to develop an HTML parser in this article, but in the October issue Paul Warren showed us how to extend `TParser` to parse HTML files (among other things), and it is possible to have too much of a good thing. Therefore, I have put the parser development part of the article into an HTML file which is included on the cover disk, for

those who are interested. The parser I developed is still included in the source code as well. Strictly speaking, `TParser` and its descendants are sophisticated lexical analysers, and an interesting project would be to replace my primitive lexical analyser with Paul's much nicer one. However, as my old maths books used to say, that is left as an exercise for the reader.

One more thing on this subject and we will get on to building the application. The standard text on writing parsers and compilers is *Compilers. Principles, Techniques and Tools* by Aho, Sethi and Ullman. The book is so well known it has a nickname: *The Dragon Book*. This is because there is a picture of a dragon on the cover. Who says programmers aren't creative? If you ever need to write a parser or compiler, you don't need to work out how to do it yourself. The area is well researched and excellent techniques are well known. All you need to do is apply them.

On to the application. For those not familiar with HTML, a brief description is that it is plain text, with a number of *tags* that control how the document is displayed. A tag has the form :

```
<TAGNAME ATTRIBUTE1=value1
  ATTRIBUTE2=value2  ...>
```

where `TAGNAME` is one of a defined set of tags (think of them as display instructions). The attributes are like procedure parameters, controlling various options depending on the tag. There are a *lot* of tags, and not all of them are supported by all browsers, but fortunately we can ignore nearly all of them. Table 1 shows the tags and associated attributes we need.

Note that we are ignoring the `APPLET`, `BLOCKQUOTE`, `DEL`, `DIV`, `EMBED`, `HEAD`, `IFRAME`, `ILAYER`, `INPUT`, `INS`, `LAYER`, `META`, `OBJECT`, `Q`, `SCRIPT`, `SPAN` and `STYLE` tags, all of which can have a URL as one of the attributes. You may wish to add these.

Tag	Attribute
A	The HREF attribute specifies the URL of a hypertext link
AREA	The HREF attribute specifies the URL of a hypertext link
BASE	This HREF attribute will modify any other URL on the page
FRAME	The SRC attribute specifies the URL of a hypertext link
IMG	The SRC attribute specifies the URL of a hypertext link, usually an image or video clip
LINK	The HREF attribute specifies the URL of another document that can be used in different ways in the current page. For example, if the REL attribute is set to STYLESHEET, the linked document is a style sheet. A style sheet is just a text file with instructions on how to modify some default characteristics of an HTML page. It is usually used to give a consistent look to all pages on a website
TITLE and /TITLE	If it exists, we will use the document title as the name of the current page on the diagram. It is simple to parse, as the text between the start and end tags cannot contain any other HTML

➤ *Table 1*

About Time Too

At long last, we can put everything together into a little application. It is based on the Http demo program that comes with Delphi 3. The list of parent pages is stored in `FParentUrlList`, an instance of a string list. It is used to navigate back through the website. Also, every time the current page changes, all existing diagram elements are deleted, except the one we just double clicked on (because deleting a component while we are executing one of its event handlers will raise an exception when we exit the handler, not surprisingly). Lastly, the method `ParseDoc` does most of the work.

The source is on the disk, but basically, what we need to do is this. We create and free an instance of my parser each time the procedure is called. We call the `Parse` method and, if no exceptions are raised, the diagram is cleared of all components (except the one just double clicked on, if any), and a new one is drawn. Initially, bitmap shapes are created for the current page and, if necessary, the immediate parent page. If there is a parent page, it is connected to the current page by creating one of the single headed arrow components we defined earlier. The processes of creating these components are encapsulated in `CreateBitmapShape` and `ConnectShapes`, because we will be performing these actions repeatedly.

We then step through the symbol table entries generated by the parser. If the entry is a title, the appropriate label caption is updated with the title of the current page. If it is a BASE tag entry, the value is stored, so that the complete URL of other elements can be calculated. If the BASE tag exists on a page, all URLs are relative to the value of its HREF attribute. If the symbol table entry is an image, the base path is added to the beginning of the URL, and a new bitmap shape is created and connected to the current page. The difference in creating this new shape is that a different image is used to visually denote an image URL. If the entry is a link, then the URL is examined to determine its type, and an image selected from a highly original collection (I'm thinking of copyrighting them). The `OnDblClick` event has an event handler assigned if the link is to another HTML document. This event handler gets the URL from the component's `caption` property, makes this URL the new current page, adjusts the parent URL list, and redraws the diagram. The `parent` component has a similar

event handler assigned to it. I also noticed that URLs that contain spaces have the characters %20 embedded in them where the spaces are supposed to go. There is therefore a routine that replaces this character string with a space.

Hip And Funky Dude

And that's it. It is a simple application, and HTML is an evolving 'standard', so you may find web pages that will not parse. You could extend the application by dealing with other tags, particularly the <Q> tag, which allows quoted text (a URL as text, not as a link, for example). You could show thumbnail sketches for the images, and add or modify event handlers so that pages and images could be edited. You could extend the error handling in the parser to recover from an error and keep processing, or use a more sophisticated positioning technique for the diagram elements.

You could also adapt the application to show idea maps, instead of websites. An idea is represented by some text, a file, an image etc. An idea map then connects related ideas. You would need some sort of data storage, but the presentation elements all exist.

This leads on to the next article, when we make the components movable, sizeable and storeable. Because I'm a trendy dude who's hip to the latest jive, the example project will be a use case editor. How's that for funky?

Jim Cooper works at Sybiz Software in Newbury, UK. He's an Australian, but we won't hold it against him, just as long as he donesn't mention the cricket (Ok, Jim?). You can email him at jim.cooper@virgin.net