# Create Complex Graphs with GraphViz

*Tamar E. Granor*
*Tomorrow's Solutions, LLC*
*Website: www.tomorrowssolutionsllc.com*
*Email: tamar@tomorrowssolutionsllc.com*

*GraphViz is an open-source tool that creates graphs in a variety of formats. It can be used for all kinds of networks, including organization charts, data structures, flow charts, family trees, and more. It offers a tremendous amount of customization.*

*In this session, we'll see how to take advantage of GraphViz from VFP, including wrapper classes to handle the messy parts.*

## Introduction

For several years, I've been helping convert a vertical market application for produce distributors from FoxPro for Windows to Visual FoxPro. A couple of years ago, the client asked me to look at how we could provide a graphical representation showing what happened to a particular lot once it arrived: how it was broken into smaller quantities or used to create other kinds of items, and then where those results were sold.

My client had recommendations for a couple of applications to do the graphing. He'd already done some preliminary looking and thought GraphViz was the way to go. He wanted me to confirm that and then make it work.

Given an example of what he wanted, it took me about 40 minutes to write very basic code for that specific case. In the process, I got a feel for how GraphViz works and what kind of options it would offer.

Using that example, I built wrapper classes to make it easy to use GraphViz for this task and others.

**Figure 1** shows a graph created by my client's application. The different shapes and colors represent different types of transactions in the application. For example, the bright green triangle at the left is the original purchase order on which a load of cantaloupes was ordered, while the yellow ovals are work orders showing whole cantaloupes being converted to chunks and culls (leftover bits discarded), and the chunks being used to make fruit salad.

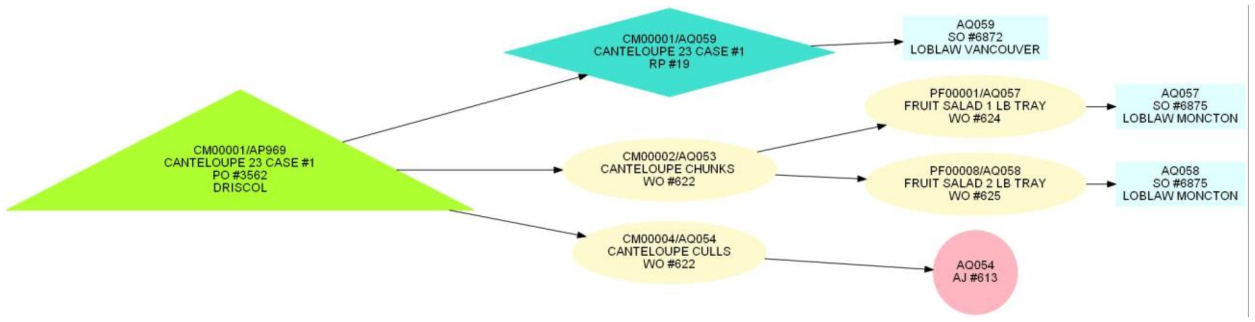


**Figure 1**. This graph was produced using GraphViz on data from a vertical market application for produce distributors.

This paper provides an overview of GraphViz and then focuses on how to use it in VFP. I'll show some of the options it provides for customizing your graph.

## What is GraphViz?

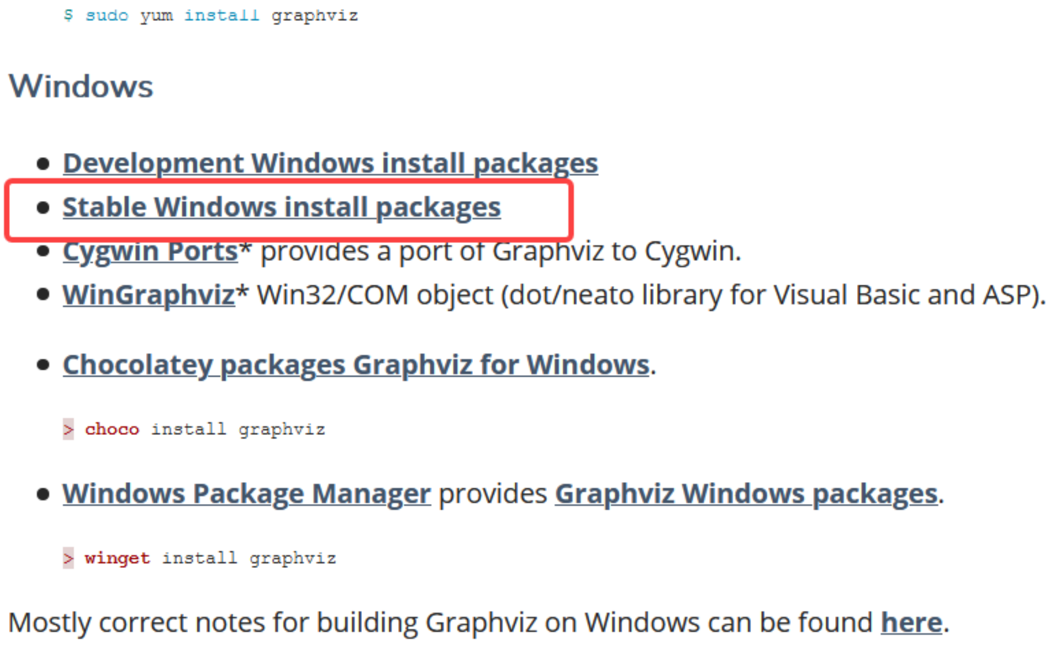
GraphViz is an open source project to produce what the creators call "graph visualizations." The word "graph" here is used in its mathematical sense, but fortunately, you don't have to know too much about graphs in math to be able to use GraphViz.

The project's home is http://graphviz.org/. It offers several different "engines" to take a text description of a graph and create the visualization. In this paper, I'll work with only one of those engines, named "dot," described as providing "'hierarchical' or layered drawings of directed graphs." What this means is that it can produce graphs with arrows, as in the earlier example.

## Setting up GraphViz

Installing and configuring GraphVia is fairly easy. (Finding the right download was harder now than when I first installed GraphViz a couple of years ago. It may change again.)

In the Windows section of the Downloads page of the GraphViz website, click "Stable Windows install packages" (see **Figure 2**). The page that appears is pretty sparse (**Figure 3**). Click "10/" and you'll see the page shown in **Figure 4**.



```
$ sudo yum install graphviz
```

Windows

- Development Windows install packages
- Stable Windows install packages
- Cygwin Ports* provides a port of Graphviz to Cygwin.
- WinGraphviz* Win32/COM object (dot/neato library for Visual Basic and ASP).
- Chocolatey packages Graphviz for Windows.

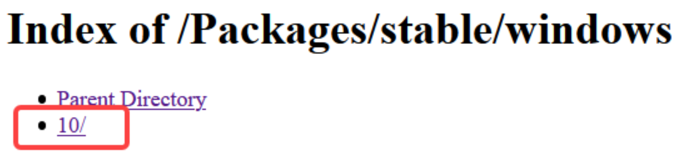
```
> choco install graphviz
```

- Windows Package Manager provides Graphviz Windows packages.

```
> winget install graphviz
```

Mostly correct notes for building Graphviz on Windows can be found here.

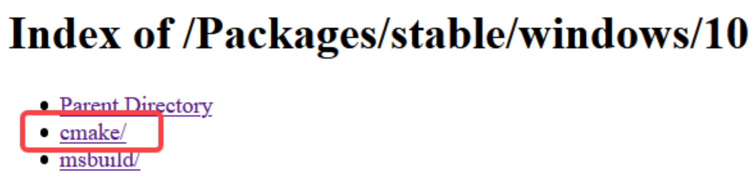
**Figure 2**. To download and install GraphViz, click the highlighted item on the Downloads page.



Index of /Packages/stable/windows

- Parent Directory
- 10/

**Figure 3**. The page for Stable Windows install packages doesn't have much on it. Click the circled item.



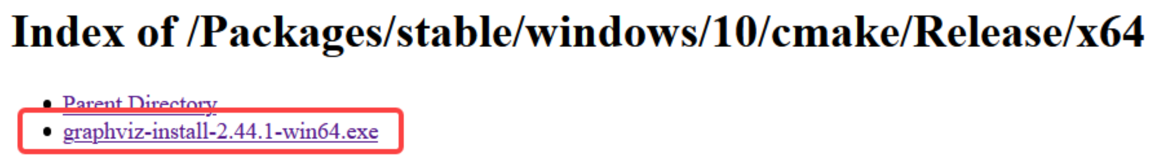
Index of /Packages/stable/windows/10

- Parent Directory
- cmake/
- msbuild/

**Figure 4**. Click the "cmake/" item on this page to continue the path to the download.

Click "cmake/" and on the next page, click "Release/". Next, click "x64/" and you'll finally land on the page that contains the download. (You can skip the one page at a time path and simply use the direct link https://www2.graphviz.org/Packages/stable/windows/10/cmake/Release/x64/.)

On this page (**Figure 5**), click the EXE to download it. The version number may be different, of course.



Index of /Packages/stable/windows/10/cmake/Release/x64

- Parent Directory
- graphviz-install-2.44.1-win64.exe

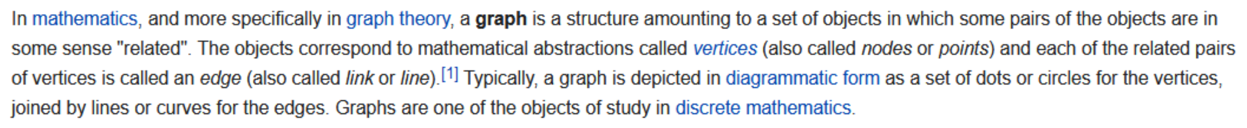
**Figure 5**. Click the EXE here to download the installation file.

Once the download is complete, double-click the file to begin installation. The examples in this session (and the downloads) assume you've accepted the default installation path of C:\Program Files\Graphviz 2.44.1, but since you'll be running GraphViz from a batch file, the exact location doesn't matter. (If you chose another location, you'll have to modify the path specified in the classes provided in the downloads for this session.)

When I installed GraphViz this way, I found that one key file was missing. Config6 is supposed to be created at installation, but it wasn't. I've included a copy in the downloads for this session. Put it in the Bin folder of your installation.

## What do we mean by "graph"?

As noted earlier, the term "graph" here is used in the mathematical sense. The definition in **Figure 6** comes from Wikipedia (https://en.wikipedia.org/wiki/Graph_(discrete_mathematics), captured August 31, 2020).



In mathematics, and more specifically in graph theory, a **graph** is a structure amounting to a set of objects in which some pairs of the objects are in some sense "related". The objects correspond to mathematical abstractions called *vertices* (also called *nodes* or *points*) and each of the related pairs of vertices is called an *edge* (also called *link* or *line*).[1] Typically, a graph is depicted in diagrammatic form as a set of dots or circles for the vertices, joined by lines or curves for the edges. Graphs are one of the objects of study in discrete mathematics.

**Figure 6.** In this session, we're talking about a specific kind of graph, the sort discussed in discrete mathematics.

The basic components of these graphs are nodes, which represent some entity; and edges, which connect nodes. In **Figure 1** earlier in this document, the nodes represent various kinds of transactions, and the edges, which are directed, show the flow through time from one transaction to another.

## Working with GraphViz

As noted earlier, GraphViz provides several different engines for generating graphs. We'll work with the engine called dot, which is provided as an EXE in the Bin folder of the GraphViz installation. dot is designed to be used from a command line. It expects a text file of instructions in a specified format and generates an output file in one or more formats. Supported output formats include a number of graphical choices as well as PDF, PostScript and quite a few others.

Creating a graph involves two steps: creating the text file with the instructions (I'll refer to that text file as the *graph description file*) and calling dot to turn those instructions into output.

### Documentation

The Documentation page of the GraphViz website includes a variety of types of documentation. The ones I've found most useful are Command-line Usage (http://graphviz.org/doc/info/command.html), which spells out how to call dot (more on this in the next section of this paper), and those describing various options, such as output formats, shapes and colors.

The documentation page also links a paper about dot, that includes examples with the data used to generate them.

The website also includes a Gallery page (http://graphviz.org/gallery/) with lots of examples of graphs. Clicking an example takes you to a page that shows the graph description file used to create it.

### Formatting data for dot

The dot engine expects data in a specific format that includes a header and then information about the nodes and edges. To make it easier to explain, we'll look at a simple example. **Figure 7** shows a very basic diagram of the employee hierarchy for the Northwind database that comes with VFP. The graph description file that produced it is shown in **Listing 1.**

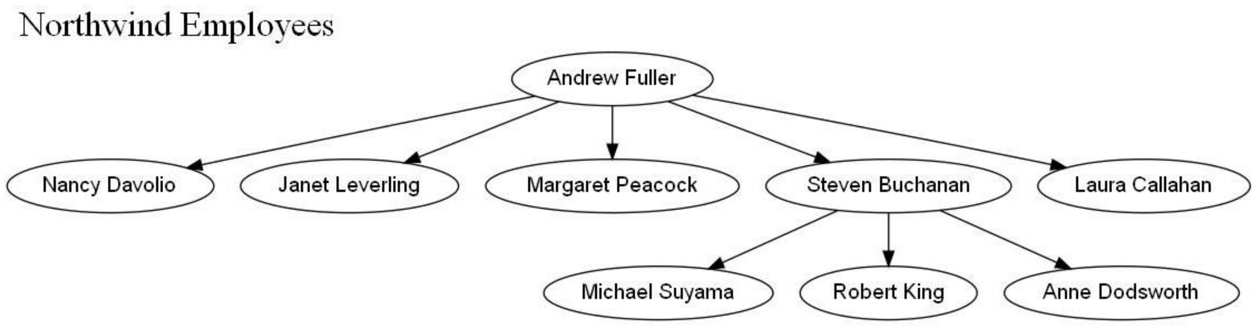


**Figure 7**. This simple graph shows the employee hierarchy for the example Northwind database.

**Listing 1**. This specification, when run through dot, produces the graph shown in **Figure 7**.

```
digraph GVFile {
  graph[rankdir=TB];
  graph[label="Northwind Employees", labelloc=top, labeljust=left, fontsize=24];
      2[label="Andrew Fuller",fontname="Arial"];
      1[label="Nancy Davolio",fontname="Arial"];
      2 ->    1;
      3[label="Janet Leverling",fontname="Arial"];
      2 ->    3;
      4[label="Margaret Peacock",fontname="Arial"];
      2 ->    4;
      5[label="Steven Buchanan",fontname="Arial"];
      2 ->    5;
      8[label="Laura Callahan",fontname="Arial"];
      2 ->    8;
      6[label="Michael Suyama",fontname="Arial"];
      5 ->    6;
      7[label="Robert King",fontname="Arial"];
      5 ->    7;
      9[label="Anne Dodsworth",fontname="Arial"];
      5 ->    9;
}
```

The header indicates that kind of graph we're creating. dot supports two kinds, known as digraphs and graphs. Digraphs are directed and use an arrowhead at (at least) one end of each edge, while graphs are undirected and do not include arrowheads. In this paper, we'll focus on digraphs. The header can also include an identifier for the graph. In **Listing 1**, the header specifies a digraph to be named GVFile.

The main body of the graph specification is enclosed in curly braces. It includes descriptions for nodes and edges and, optionally, various attributes of the graph. There are three kinds of lines in **Listing 1**. The second and third lines contain graph level information. The lines that begin with a number and are followed with square brackets describe nodes, and the lines that show two numbers separated by an arrow describe edges. Each line ends with a semi-colon, though they are optional.

The graph itself, as well as nodes and edges, can have attributes. Attributes are described inside square brackets using name-value pairs in the form AttributeName=AttributeValue. The list of attributes available is documented at http://graphviz.org/doc/info/attrs.html.

### Describing the graph itself

Our example shows only a few graph attributes. They're on the second and third lines of the file, preceded by "graph" and wrapped in square brackets.

The rankdir attribute indicates what direction the graph runs. The value "TB" shown here means top to bottom. You can also specify "LR", meaning left to right; that's the value used for the graph shown in **Figure 1**.

The third line of the example has a set of attributes used to label the graph. Other attributes for the graph as a whole include bgcolor for background color, landscape to indicate whether to use portrait (false) or landscape (true) mode, and rotation to specify a number of degrees to rotate the completed graph. See the documentation for the complete list.

### Describing nodes

Node lines begin with an identifier for the node. In **Listing 1**, we're using the primary keys from the Employee records since they're already known to be unique. The identifier is only displayed in the graph if you don't specify a label for the node.

There are lots of attributes you can specify for nodes. In this example, we specify only two: label, which indicates the text to appear for the node and fontname, which indicates what font to use for the label. We could, of course, specify fontsize (the default is 14 points) and fontcolor (default is black).

By default, the node is enclosed in an oval. We can specify a different shape and details for whatever shape we use. The details for putting a shape at a node are discussed later in this paper, in the section "Varying shapes."

### Describing edges

The example shows the simplest version of edge specification: the identifier for one node, followed by an arrow, followed by the identifier for another node. That produces an edge with an arrowhead pointing to the second node.

You can specify attributes for an edge, such as color, arrowhead (which determines what the arrowhead looks like), and a whole set about adding a label. See "Making edges more informative," later in this paper.

## Running dot

dot is an executable; it accepts parameters to tell it what graph description to work with, what kind of output to create, and where to put the output. **Listing 2** shows the command line used to create **Figure 7** from **Listing 1**. (This command was run from a batch file, where the whole thing was on a single line. It's been wrapped onto multiple lines here for readability.)

**Listing 2.** To run dot, call the EXE passing the desired output type, the input file name, and the output file name.

```
"C:\Program Files\Graphviz 2.44.1\bin\dot.exe" -Tjpeg
C:\USERS\TAMAR\APPDATA\LOCAL\TEMP\20200904110422.GV
-o C:\USERS\TAMAR\APPDATA\LOCAL\TEMP\20200904110422.jpeg
```

The -T flag indicates the desired output type. See http://graphviz.org/doc/info/output.html for a list of choices; it includes multiple graphic formats, PDF, PostScript, and more.

The input file is specified without a flag and is the only item specified that way.

The -o flag specifies the name of the output file. Flags are case-sensitive, so "-o" is not the same as "-O" (which indicates that output file names should be generated automatically from the input file name and output file type).

## Using GraphViz from VFP

To simplify creating graphs from my client's application, I created a set of classes that make it easy to build the graph description and then to run it through dot.

### A generic GraphViz class

The top-level class, GVGraph (GVGraph.PRG in the downloads for this session) sets up the properties and methods needed for both parts of graph creation. Some are abstract at this level, while others contain code. The principal method, meant to be called from outside the class, is MakeGraph, which constructs the graph description file and then generates the graph; the code is shown in **Listing 3**.

**Listing 3.** This is the principal method in the GVGraph class. It manages the process of generating both the graph description file and the graph itself.

```
PROCEDURE MakeGraph(tcDataCursor, tcFormat)

This.cErrorMessages = ''
This.Setup()
This.CreateColorShapeCursor()
This.LoadColorsAndShapes()

This.SpecifyGraphData(m.tcDataCursor)
IF NOT EMPTY(m.tcFormat) AND VARTYPE(m.tcFormat) == "C"
   This.SetGraphFormat(m.tcFormat)
ENDIF

IF EMPTY(This.cErrorMessages)
   This.GenerateGVFile()
ENDIF

IF EMPTY(This.cErrorMessages)
   This.GenerateGraph()
ENDIF
```

```
RETURN
ENDPROC
```

MakeGraph accepts two optional parameters. The first, tcCursor, is the alias for a cursor containing the data to be graphed. The second parameter to MakeGraph, tcFormat, specifies the desired output format.

MakeGraph starts with a bunch of set-up work. These classes manage errors by adding data to the cErrorMessages property, which can be interrogated at any point, including after the process is ostensibly complete. MakeGraph checks for errors at each major step and proceeds only if there are none, so the first step is to clear that property to ensure no error messages are left from a previous call.

The Setup method is abstract in this class, provided to allow any work that needs to be done before generating the graph, such as setting any default values. We'll look at how it might be used in subclasses.

As noted earlier, GraphViz allows colors and shapes to be specified for nodes. My client wanted the ability to link colors and shapes with transaction types, so I designed a system that uses a cursor to hold a look-up table. The CreateColorShapeCursor method creates and indexes the cursor. By creating the cursor here, I could ensure that it has the expected structure, but filling it is a job for a subclass designed for a specify type of graph, so the LoadColorsAndShapes method is abstract here.

In most situations however, you're more likely to want to use shape and color to show two different characteristics, so in this version of the class, CreateColorShapeCursor creates two cursors, one for colors and one for shapes.

The class supports two approaches to specifying the data to be graph, either passing in a cursor or running code as part of the graph creation process. The SpecifyGraphData method (shown in **Listing 4**) called from MakeGraph calls either SetGraphData or GetGraphData, depending on whether the tcCursor parameter is character. SetGraphData simply sets the class's cDataCursor property to the value passed in. GetGraphData is abstract here; to use it, you need to put code into the method in a subclass.

**Listing 4**. The SpecifyGraphData method lets the class support two different approaches to supplying the data to be graphed.

```
PROCEDURE SpecifyGraphData(tcCursor)
* Provide data for the graph. Can call either
* SetGraphData to point to an existing cursor
* or GetGraphData to do the work here.
* We'll decide based on whether the param is provided.

LOCAL llReturn

llReturn = .t.
IF VARTYPE(m.tcCursor) = 'C'
```

```
   llReturn = This.SetGraphData(m.tcCursor)
ELSE
   llReturn = This.GetGraphData()
ENDIF

RETURN
```

Next, if the tcFormat property has a non-empty, character value, the SetGraphFormat method stores that value in the class's cGraphFormat property. This class has no default value for this property. It's a good idea to specify one in a subclass; we'll see an example in "Graphing Northwind Employees," later in this paper.

With all that set-up done, we're ready to generate the graph description file; that's the job of the GenerateGVFile method, shown in **Listing 5**.

**Listing 5**. The GenerateGVFile method traverses the specified data to create the graph description file.

```
PROCEDURE GenerateGVFile
* Generate the GV file to be used for the graph

LOCAL llContinue

llContinue = .T.

* Check we have all we need
IF EMPTY(This.cDataCursor)
   This.AddError("Must specify name of data cursor.")
   llContinue = .F.

   IF m.llContinue
      IF NOT USED(This.cDataCursor)
         This.AddError("Data cursor must be open.")
         llContinue = .F.
      ENDIF
   ENDIF

ENDIF

IF m.llContinue
   IF EMPTY(This.cGVFile)
      This.GenerateGVFileName()
   ENDIF
ELSE
   RETURN .F. && bail out early
ENDIF


LOCAL lnOldSelect

lnOldSelect = SELECT()
SELECT (This.cDataCursor)

This.GenerateGVHeader()
```

```
SCAN
   lcNodeName = This.GenerateGVNode()
   This.GenerateGVNodeEdges(m.lcNodeName)
ENDSCAN

This.GenerateGraphFooter()

SELECT (m.lnOldSelect)

RETURN
ENDPROC
```

After confirming a data cursor has been specified and is open, GenerateGVFile ensures there's a name for the graph description file. If none has been specified, it calls GenerateGVFileName, which creates a file name based on the current date and time, with an extension of GV, using the temporary folder as the path. The code is shown in **Listing 6**.

**Listing 6**. If no file name is specified for the graph description file, this method creates one.

```
PROCEDURE GenerateGVFileName
* Generate a name for the GV file that contains the graph definition

LOCAL lcFileName

lcFileName = TTOC(DATETIME(), 1)
lcFileName = FORCEEXT(FORCEPATH(m.lcFileName,SYS(2023)), 'GV')

This.cGVFile = m.lcFileName

RETURN
ENDPROC
```

Once all that housekeeping is handled, GenerateGVFile begins the actual generation process. All the methods called in this section are abstract, because the details of what needs to be generated vary with both the graph type and the data.

First, GenerateGVHeader is called to generate the opening part of the file.

Next, we loop through the records in the data cursor. For each, we call GenerateGVNode to create a node line and GenerateGVNodeEdges to create any edges involving that node. (This was a design choice that makes sense for my client's application. For some situations, it might make sense to process the cursor once to create all nodes, and then again to add the edges.)

Finally, GenerateGVFooter is called to create anything needed at the end of the file.

If GenerateGVFile is successful, the final section of MakeGraph calls GenerateGraph to convert the graph definition to the desired output format. GenerateGraph, shown in **Listing 7**, does this by creating and running a batch file.

**Listing 7**. The GenerateGraph method constructs a batch file that runs the specified GraphViz graphing engine to convert the graph description to a graph in the specified format.

```
PROCEDURE GenerateGraph
* Generate the graph from the GV file

LOCAL llContinue, lcGraphCall

llContinue = .T.

* Check conditions
IF EMPTY(This.cGVPath)
   This.AddError("GraphViz path must be specified.")
   llContinue = .F.
ENDIF

IF m.llContinue
   IF EMPTY(This.cGVFile)
      This.AddError("GV file must be specified.")
      llContinue = .F.
   ENDIF

   IF m.llContinue
      IF NOT FILE(This.cGVFile)
         This.AddError("GV file must be created before generating graph.")
         llContinue = .F.
      ENDIF
   ENDIF
ENDIF

IF m.llContinue
   LOCAL lcExt, lcFileName

   DO CASE
   CASE EMPTY(This.cGraphFile)
      IF EMPTY(This.cGraphFormat)
         lcExt = "JPEG"
      ELSE
         lcExt = This.cGraphFormat
      ENDIF

      This.cGraphFile = FORCEEXT(This.cGVFile, m.lcExt)

   CASE EMPTY(JUSTEXT(This.cGraphFile))
      * Add extension
      IF EMPTY(This.cGraphFormat)
         lcExt = "JPEG"
      ELSE
         lcExt = This.cGraphFormat
      ENDIF

      This.cGraphFile = FORCEEXT(This.cGraphFile, m.lcExt)

   ENDCASE
```

```
ELSE

   RETURN .F. && early exit
ENDIF

WAIT WINDOW NOWAIT 'Please wait, generating graph'

LOCAL lcBatFile
lcBatFile = This.CreateBatFile()

*-- Determine the call command to generate the graph
lcGraphCall = This.GenerateGraphCall()

STRTOFILE(m.lcGraphCall + CHR(13) + CHR(10), m.lcBatFile, .f.)
STRTOFILE("timeout /t 1", m.lcBatFile, .t.)

LOCAL llError

llError = This.RunBatFile(m.lcBatFile)

IF m.llError
   This.AddError("Error occurred while running batch file.")
ENDIF
WAIT CLEAR

RETURN
ENDPROC
```

After confirming that the path to the graphing engine has been specified and that the graph definition file exists, the method confirms there's an output type specified, using JPEG as the default. It then constructs a name for the graph file.

If all that is successful, the CreateBatFile method is called to generate a name for a batch file. In this case, the batch file is named makgraph.bat and is placed in the user's temporary folder, as shown in **Listing 8**.

**Listing 8**. By default, the batch file is named makgraph.bat and stored in the temporary folder.

```
PROCEDURE CreateBatFile
* Created method to abstract this

*-- create a variable for the batfile, and give it a value
LOCAL lcBatFile
lcBatFile = FORCEPATH("makgraph.bat", SYS(2023))

RETURN m.lcBatFile
ENDPROC
```

Next, the GenerateGraphCall method constructs the necessary call to the graphing engine. That method is abstract in this class, as the details vary for the different graphing engines. The returned string is written to the batch file, followed by a timeout command to ensure the batch file has time to complete the task before the batch file ends.

With the batch file completed, the next step is to run it; that's what the RunBatFile method, shown in **Listing 9** does. The error handling here is for the case where the RUN command is unable to find or to run the batch file. It doesn't catch cases where the batch file itself fails.

**Listing 9**. The RUN command is used to execute the batch file that creates the graph.

```
PROCEDURE RunBatFile(lcBatFile)
* Run the specified batch file

*-- Store the setting of "on error", and then override it
LOCAL lcSavError
lcSavError = on("error")

LOCAL llError
llError = .F.

ON ERROR llError = .T.

RUN /n7 &lcBatFile

*-- restore the value of "on error"
ON ERROR &lcSavError


RETURN m.llError
ENDPROC
```

The original version of the class had a call to the DeleteBatFile method after the call to RunBatFile, but too often, that method ran too soon and caused problems. So this version of the code omits the call. You could use VFP's WAIT command or the Windows API Sleep function to ensure the batch file has finished before running it, if deleting the batch file is critical.

Finally, if an error occurred in running the batch file, we add that information to the error log. The AddError method is shown in **Listing 10**.

**Listing 10**. The AddError method adds a new message to the cErrorMessages property.

```
PROCEDURE AddError(tcMessage)
* Add an error message to the error property

This.cErrorMessages = This.cErrorMessages + ALLTRIM(m.tcMessage) + CHR(13) + CHR(10)


RETURN
ENDPROC
```

GVGraph has a number of methods, shown in **Table 1**, designed to be used by the GenerateGVNode and GenerateGVNodeEdges methods. They provide information used in constructing those lines.

**Table 1**. These methods are designed to be called by the GenerateGVNode and GenerateGVNodeEdges methods.

| **Method** | **Purpose** | **Abstract?** |
|---|---|---|
| GetNodeColor | Returns the color for the specified node type, by looking it up in the color cursor. | |
| GetNodeID | Returns a unique ID for the specified node. | Yes |
| GetNodeLabel | Returns a label for the specified node. | Yes |
| GetNodeShape | Returns the shape for the specified node type, by looking it up in the shape cursor. | |
| WriteGraphLine | Saves the specified line of the graph to the graph description file. | |

The final set of methods is for displaying the graph. ShowGraph, shown in **Listing 11**, manages the process of displaying the graph. After confirming the graph exists (including waiting up to three seconds for the file to finish being written), it calls ShowGraphNative to do the actual display.

**Listing 11**. ShowGraph manages the process of displaying the graph once it's created.

```
PROCEDURE ShowGraph
* Display the generated graph.
* Parameter indicates where to display it

LOCAL llProceed

llProceed = .T.

IF EMPTY(This.cGraphFile)
   This.AddError("Graph file not specified. Can't display graph.")
   llProceed = .F.
ENDIF

IF m.llProceed
   * Allow some time in case file hasn't been written yet.
   lnStart = SECONDS()
   DO WHILE NOT FILE(This.cGraphFile) AND SECONDS() - m.lnStart < 3
   ENDDO

   IF NOT FILE(This.cGraphFile)
      This.AddError("Graph file doesn't exist. Can't display.")
      llProceed = .F.
   ENDIF
ENDIF

IF NOT m.llProceed
   RETURN .F. && early exit
ENDIF

This.ShowGraphNative()

RETURN
ENDPROC
```

ShowGraphNative, shown in **Listing 12**, uses a routine called ExecuteFile, created by Doug Hennig and documented in his paper at https://doughennig.com/papers/Pub/E-WAPI.pdf, to display the graph using whatever application Windows is set to for the specified file type. ExecuteFile is a wrapper around the ShellExecute API function, and it returns the success value. Anything under 32 means failure.

**Listing 12**. ShowGraphNative displays the graph using the "native" application for the file type.

```
PROCEDURE ShowGraphNative
* Show the newly generated graph in its native app

* If we get here (assuming we came in the right way), the file exists
* Use ShellExecute to "run" the file via Doug's wrapper

* handle failure
LOCAL lnSuccess
lnSuccess = ExecuteFile(This.cGraphFile,'open')
IF m.lnSuccess < 32
   This.AddError("Unable to display graph. Error # " + TRANSFORM(m.lnSuccess)
ENDIF m.lnSuccess < 32

RETURN
ENDPROC
```

My client's application maintains a list of preferred applications for various file types, so it includes an alternative method for displaying the graphs and has code in ShowGraph to decide which method to use.

## A dot-specific subclass

The second class in my class hierarchy is GVDotGraph (gvdotgraph.prg in the downloads for this session), which sets up the dot-specific aspects of generating graphs.

The Setup method adds one line, shown in **Listing 13**, to provide the path to the dot executable. Here, it's installed in the default location. If you've installed it elsewhere, you'll need to modify this line.

**Listing 13**. This line, added to the Setup method, points to the dot executable.

```
This.SetGVPath("C:\Program Files\Graphviz 2.44.1\bin\dot.exe")
```

Three methods that are abstract in GVGraph are populated here. Since this class knows it's using dot, GenerateGraphCall can build the command-line instruction to execute it. The code is shown in **Listing 14**.

**Listing 14**. GenerateGraphCall constructs the line needed to run dot, passing in the graph description file and generating output in the specified format.

```
PROCEDURE GenerateGraphCall
* Set up the call to DOT based on properties
LOCAL lcGraphCall
```

```
lcGraphCall = '"' + This.cGVPath + '" -T' + ALLTRIM(This.cGraphFormat)
lcGraphCall = m.lcGraphCall + ' ' + ALLTRIM(This.cGVFile)
lcGraphCall = m.lcGraphCall + ' -o ' + ALLTRIM(This.cGraphFile)

RETURN m.lcGraphCall
ENDPROC
```

In addition, knowing we're using dot and having decided we're building only digraphs, the GenerateGVHeader (**Listing 15**) and GenerateGVFooter (**Listing 16**) methods contain the necessary code to create those parts of the graph definition file.

**Listing 15**. Knowing we're using dot, we can construct the header portion of the graph description file, specifying the type of graph, its orientation, and title information.

```
PROCEDURE GenerateGVHeader
* Put the header info into the GV file

LOCAL lcLine

lcLine = 'digraph GVFile {' + CHR(13) + CHR(10)
* First line starts a new file
This.WriteGraphLine(m.lcLine, .T.)

lcLine = '  graph[rankdir=' + This.cGraphOrientation + '];' + CHR(13) + CHR(10)
This.WriteGraphLine(m.lcLine)

IF NOT EMPTY(This.cGraphTitle)
   lcLine = '  graph[label="' + ;
            This.cGraphTitle + '", labelloc=top, labeljust=left, fontsize=24];' + ;
            CHR(13) + CHR(10)
   This.WriteGraphLine(m.lcLine)
ENDIF

RETURN
ENDPROC
```

**Listing 16**. The footer for a dot graph is simply a closing curly brace.

```
PROCEDURE GenerateGraphFooter
* Just close the curly brace

This.WriteGraphLine('}')

RETURN
ENDPROC
```

This is as much as we can do generically. Everything else is specific to the graph we're constructing.

## Graphing Northwind Employees

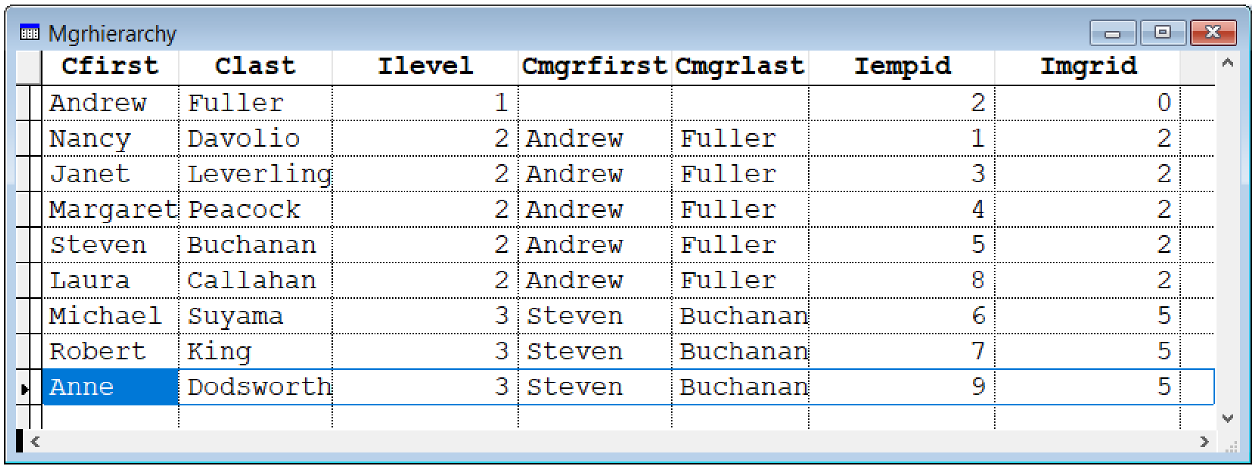
In order to create any specific graph, you need to collect the data to be graphed and get it into a form that makes it possible (even better, easy) to generate the graph definition file, and a subclass of GVDotGraph that actually does the work.

As noted earlier, you can collect the data outside the graphing class or you can put the code to collect the data into the GetGraphData method. For this first example, I've chosen to do it separately and pass the cursor name into MakeGraph. The code to do so is shown in **Listing 17** and included in the downloads for this session as NWEmps.PRG.

I've written previously about working with hierarchical data like this, so I won't go into detail here. If you want the background on how this code works, see the section "Who does an employee manage?" in http://www.tomorrowssolutionsllc.com/Articles/Handling%20Hierarchical%20Data.pdf; the code here is based on MgrHierarchy.PRG described there. When it's done, the cursor MgrHierarchy contains each Northwind employee with the name and ID of their manager, as shown in **Figure 8**; it's ordered with the big boss first, then his direct reports, and then their direct reports, and so on. (In fact, Northwind has only a three-level hierarchy, but this code works no matter how many levels there are.)

**Listing 17**. This code drills down the Northwind management hierarchy and builds a cursor showing each employee with information about their manager.

```
* Build the employee hierarchy from top to bottom
LOCAL iCurrentID, iLevel, cFirst, cLast, nCurRecNo, cMgrFirst, cMgrLast

OPEN DATABASE HOME(2) + "Northwind\Northwind"

CREATE CURSOR MgrHierarchy ;
   (cFirst C(15), cLast C(20), iLevel I, ;
    cMgrFirst C(15), cMgrLast C(15), iEmpID I, iMgrID I)
CREATE CURSOR EmpsToProcess ;
   (EmployeeID I, cFirst C(15), cLast C(20), iLevel I, ;
    cMgrFirst C(15), cMgrLast C(15), ReportsTo I)

INSERT INTO EmpsToProcess ;
   SELECT EmployeeID, FirstName, LastName, 1, "", "", ReportsTo ;
      FROM Employees ;
      WHERE ReportsTo = 0

SELECT EmpsToProcess

SCAN
   iCurrentID = EmpsToProcess.EmployeeID
   iLevel = EmpsToProcess.iLevel
   cFirst = EmpsToProcess.cFirst
   cLast = EmpsToProcess.cLast
   cMgrFirst = EmpsToProcess.cMgrFirst
   cMgrLast = EmpsToProcess.cMgrLast
   iReportsTo = EmpsToProcess.ReportsTo
   * Insert this records into result
   INSERT INTO MgrHierarchy ;
      VALUES (m.cFirst, m.cLast, m.iLevel, ;
              m.cMgrFirst, m.cMgrLast, m.iCurrentID, m.iReportsTo)

   * Grab the current record pointer
   nCurRecNo = RECNO("EmpsToProcess")

   INSERT INTO EmpsToProcess ;
      SELECT EmployeeID, FirstName, LastName, m.iLevel + 1, ;
             m.cFirst, m.cLast, ReportsTo ;
         FROM Employees ;
         WHERE ReportsTo = m.iCurrentID

   * Restore record pointer
   GO m.nCurRecNo IN EmpsToProcess
ENDSCAN

SELECT MgrHierarchy
```


Mgrhierarchy

| Cfirst | Clast | Ilevel | Cmgrfirst | Cmgrlast | Iempid | Imgrid |
|---|---|---|---|---|---|---|
| Andrew | Fuller | 1 | | | 2 | 0 |
| Nancy | Davolio | 2 | Andrew | Fuller | 1 | 2 |
| Janet | Leverling | 2 | Andrew | Fuller | 3 | 2 |
| Margaret | Peacock | 2 | Andrew | Fuller | 4 | 2 |
| Steven | Buchanan | 2 | Andrew | Fuller | 5 | 2 |
| Laura | Callahan | 2 | Andrew | Fuller | 8 | 2 |
| Michael | Suyama | 3 | Steven | Buchanan | 6 | 5 |
| Robert | King | 3 | Steven | Buchanan | 7 | 5 |
| Anne | Dodsworth | 3 | Steven | Buchanan | 9 | 5 |

**Figure 8**. This cursor, created by the code in **Listing 17**, has one record for each Northwind employee, with information about the employee and that person's manager.

The class GraphNWEmp1 was used to create the simple graph in **Figure 7**; it's included in the downloads for this session as gvnwempgraph1.prg. It contains code in only five methods.

The least interesting of these is the Setup method, which is extended to specify JPEG for output, top-to-bottom orientation and a title of "Northwind Employees." It's shown in **Listing 18**.

**Listing 18**. The Setup method of GraphNWEmp1 specifies a few characteristics of the graph.

```
PROCEDURE Setup
* Specify format and orientation

DODEFAULT()
IF EMPTY(This.cGraphFormat)
```

```
   * Specify format if we didn't already
   This.SetGraphFormat('jpeg')
ENDIF

* Set top to bottom orientation. (Other option is 'LR')
This.SetGraphOrientation('TB')

This.SetGraphTitle('Northwind Employees')

RETURN
ENDPROC
```

Conversely, the most interesting of these methods is GenerateGVNode, shown in **Listing 19**,  which creates the line needed to specify a node.

**Listing 19**. The code in GenerateGVNode adds a line to the graph definition file for the current record.

```
PROCEDURE GenerateGVNode
* Generate GV line to create a single node,
* based on the current line of the cursor

LOCAL lcNodeDef, lcNodeID, lcNodeLabel

* Make sure we're in the right workarea
LOCAL lnOldSelect

lnOldSelect = SELECT()
SELECT (This.cDataCursor)

lcNodeID = This.GetNodeID(iEmpID)
lcNodeLabel = This.GetNodeLabel()

lcNodeDef = "   " + m.lcNodeID + ;
            '[label="' + m.lcNodeLabel + '",fontname="' + This.cFontName + '"];' + ;
            CHR(13) + CHR(10)
This.WriteGraphLine(m.lcNodeDef)

SELECT (m.lnOldSelect)

RETURN m.lcNodeID

RETURN
ENDPROC
```

The GetNodeID and GetNodeLabel methods construct an ID value and the label we want to display for each node. GetNodeID, shown in **Listing 22**, simply converts an integer employee ID to character. Because it might be used for a record other than the current one in the data cursor, it accepts the relevant employee ID as a parameter.

**Listing 20**. The GetNodeID method accepts an employee ID and converts it to a character ID value used to uniquely identify the corresponding node in the graph.

```
PROCEDURE GetNodeID(lnEmpID)
```

```
LOCAL lcEmpID

lcEmpID = PADL(lnEmpID, 4)

RETURN m.lcEmpID
ENDPROC
```

GetNodeLabel (**Listing 21**) concatenates the employee's name from the current record to form the desired label to appear on the node.

**Listing 21**. For this graph, GetNodeLabel concatenates the employee's name.

```
PROCEDURE GetNodeLabel
* Use the employee name as the label
LOCAL lcLabel

lcLabel = ALLTRIM(cFirst - (' ' + cLast))

RETURN m.lcLabel
ENDPROC
```

In this graph, each node can be the target of only a single edge, the one that comes from that employee's manager. The GenerateGVNodeEdges method (**Listing 22**) accepts the node's ID and adds a line to the graph definition file to create that edge. Note that the order of the original data ensures that the manager's node has already been defined before we generate the edge. (In general, the code to define a node should appear before any edge definitions referencing that node.)

**Listing 22**. In this example, there's only one edge coming to a particular node.

```
PROCEDURE GenerateGVNodeEdges(tcNodeID)
* Generate the edges for the specified node

LOCAL lcParentID, lcChildID
LOCAL lcEdgeLine

LOCAL lnOldSelect
lnOldSelect = SELECT()
SELECT (This.cDataCursor)

IF NOT EMPTY(iMgrID)
   lcParentID = This.GetNodeID(iMgrID)

   lcEdgeLine = "   " + m.lcParentID + " -> " + m.tcNodeID + ";" + CHR(13) + CHR(10)
   This.WriteGraphLine(m.lcEdgeLine)
ENDIF

SELECT (m.lnOldSelect)

RETURN
ENDPROC
```

In some cases, a newly added node might be the target of multiple edges in the graph. In such cases, the GenerateGVEdges method might contain a loop.

### Running the graph

With all the pieces in places, all that's left is to put them together. In this case, that means running the code to set up the data, instantiating the specific class and calling the MakeGraph method. If we want, we can display the graph right away by calling the ShowGraph method. The program in **Listing 23**, included in the downloads for this session as RunGraphNWEmp1.PRG, does all that.

**Listing 23**. It takes only a little code to create the graph, once all the components have been built.

```
* Northwind Employee graph, version 1 (simple)

* First, collect the data
DO NWEmps.PRG

* Now create the graph
LOCAL loGenerateGraph
loGenerateGraph = NEWOBJECT("GraphNWEmp1", "gvNWEmpGraph1.prg")

IF VARTYPE(loGenerateGraph) = 'O' AND NOT ISNULL(m.loGenerateGraph)
   loGenerateGraph.MakeGraph('mgrhierarchy','jpeg')
   IF EMPTY(loGenerateGraph.cErrorMessages)
      loGenerateGraph.ShowGraph()
   ELSE
      MESSAGEBOX(loGenerateGraph.cErrorMessages)
   ENDIF
ENDIF

RETURN
```

## Enhancing nodes

The graph in **Figure 7** is basic. There are lots of things we can do to make it more interesting. In this section, we'll look at options that make graphs more attractive and informative.

### Varying shapes

While it's not particularly meaningful in this example, in some cases, using different shapes for different types of nodes can be useful. The graph shown in **Figure 9** chooses a shape for each node, based on the level in the organizational chart. The class to create it is GraphNWEmp2 and it's in gvNWEmpGraph2.PRG in the downloads for this session. The downloads also include RunGraphNWEmp2.PRG to instantiate the class and create the graph.

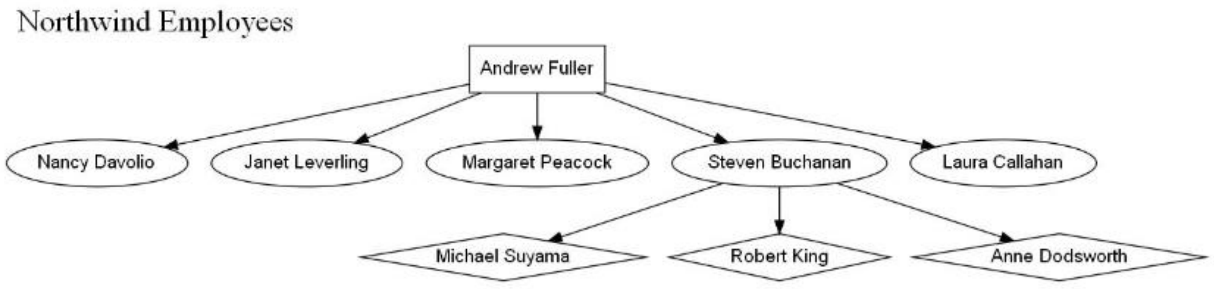


**Figure 9**. In this version of the graph, nodes at different levels in the organizational structure use different shapes.

It takes only a little additional code to get from the previous example to this one. First, we need to populate the shape cursor in the LoadColorsAndShapes method. Here (**Listing 24**), the list is hard-coded, but you could populate the cursor from a table or a text file, if you want.

**Listing 24**. To specify different shapes for different kinds of nodes, populate the shapes cursor.

```
PROCEDURE LoadColorsAndShapes
* Populate the color/shape cursors

INSERT INTO csrShapes (TypeCode, cShape) ;
   VALUES ("1", "box")
INSERT INTO csrShapes (TypeCode, cShape) ;
   VALUES ("2", "ellipse")
INSERT INTO csrShapes (TypeCode, cShape) ;
   VALUES ("3", "diamond")

RETURN
ENDPROC
```

With shapes available, we need to modify GenerateGVNode to look up the desired shape and include it in the description of each node in the graph definition file. **Listing 25** shows the modified part of the mode.

**Listing 25**. To include shape information for each node, we look up the shape to use and then include the shape attribute in the node definition.

```
lcNodeShape = This.GetNodeShape(TRANSFORM(iLevel))

lcNodeDef = "    " + m.lcNodeID + '[shape="' + m.lcNodeShape + ;
            '",label="' + m.lcNodeLabel + ;
            '",fontname="' + This.cFontName + '"];' + CHR(13) + CHR(10)
```

The shape cursor is designed to use character strings for the node types, so the call to GetNodeShape converts the iLevel field, which contains the employee's level in the hierarchy, into a string.

**Listing 26** shows the graph definition file for this new version of the Employee hierarchy. You can see that each node definition now includes the shape attribute.

**Listing 26**. This version of the graph definition file includes the shape attribute for each node.

```
digraph GVFile {
  graph[rankdir=TB];
  graph[label="Northwind Employees", labelloc=top, labeljust=left, fontsize=24];
      2[shape="box",label="Andrew Fuller",fontname="Arial"];
      1[shape="ellipse",label="Nancy Davolio",fontname="Arial"];
      2 ->    1;
      3[shape="ellipse",label="Janet Leverling",fontname="Arial"];
      2 ->    3;
      4[shape="ellipse",label="Margaret Peacock",fontname="Arial"];
      2 ->    4;
      5[shape="ellipse",label="Steven Buchanan",fontname="Arial"];
      2 ->    5;
      8[shape="ellipse",label="Laura Callahan",fontname="Arial"];
      2 ->    8;
      6[shape="diamond",label="Michael Suyama",fontname="Arial"];
      5 ->    6;
      7[shape="diamond",label="Robert King",fontname="Arial"];
      5 ->    7;
      9[shape="diamond",label="Anne Dodsworth",fontname="Arial"];
      5 ->    9;
}
```

Once you have shapes, you can decorate them. The style attribute lets you modify the shape in a variety of ways. You can specify "filled" to fill in the background (in gray, by default, but you can specify the color) and "rounded" to round the corners of the shapes. You can also specify the type of line to use for the shape's border ("dashed", "dotted", "solid") as well as that the border should be "bold". We'll demonstrate filled shapes in the next section in conjunction with colors.

## Varying colors

Color offers another way to provide additional information in a graph. As noted earlier, you can set the font color for an element using the fontcolor attribute. For node shapes, you can set two different colors. The color attribute specifies the border color of a shape.

If you include the filled attribute, you can also specify the inside color. To have it the same as the border color, you just need the color and filled attributes. If you want the inside of a shape to use a different color than the border, specify the fillcolor attribute.

GraphViz offers a number of ways to specify colors. You can provide RGB, RGBA or HSV values, or you can specify a color name. The options for color names (which we'll use here) are documented at http://graphviz.org/doc/info/colors.html.

Let's expand the previous example to color-code nodes according to job title, as in **Figure 10**.

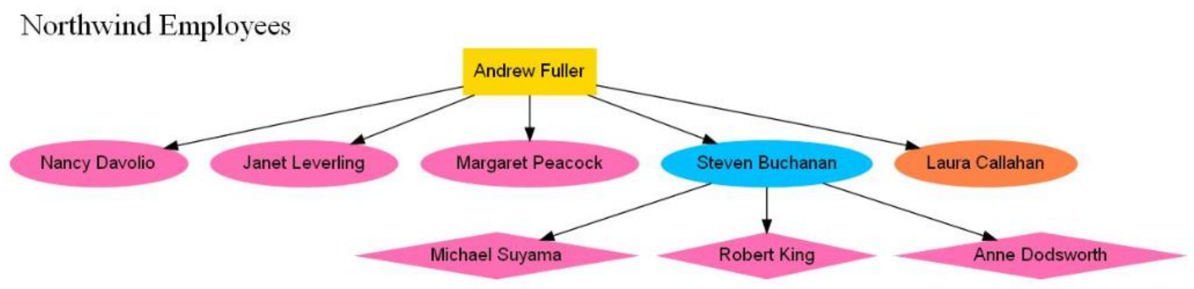


**Figure 10**. In this version of the employee hierarchy, nodes are color-code based on the job title.

First, we need to add job title information to the cursor we're using to build the graph. That requires adding that field to both cursors in NWEmps and then populating the new field. **Listing 27** shows the updated code; it's included in the downloads for this session as NWEmpsWTitle.PRG.

**Listing 27**. This code collects the employee data including job title.

```
* Build the employee hierarchy from top to bottom
LOCAL iCurrentID, iLevel, cFirst, cLast, nCurRecNo, cMgrFirst, cMgrLast
LOCAL iReportsTo, cTitle

OPEN DATABASE HOME(2) + "Northwind\Northwind"

CREATE CURSOR MgrHierarchy ( ;
   cFirst C(15), cLast C(20), iLevel I, ;
   cMgrFirst C(15), cMgrLast C(15), ;
   iEmpID I, iMgrID I, cTitle C(30) ;
  )
CREATE CURSOR EmpsToProcess ( ;
   EmployeeID I, cFirst C(15), cLast C(20), iLevel I, ;
   cMgrFirst C(15), cMgrLast C(15), ReportsTo I, cTitle C(30) ;
  )

INSERT INTO EmpsToProcess ;
   SELECT EmployeeID, FirstName, LastName, 1, "", "", ReportsTo, Title ;
      FROM Employees ;
      WHERE ReportsTo = 0

SELECT EmpsToProcess

SCAN
   iCurrentID = EmpsToProcess.EmployeeID
   iLevel = EmpsToProcess.iLevel
   cFirst = EmpsToProcess.cFirst
   cLast = EmpsToProcess.cLast
   cMgrFirst = EmpsToProcess.cMgrFirst
   cMgrLast = EmpsToProcess.cMgrLast
   iReportsTo = EmpsToProcess.ReportsTo
   cTitle = EmpsToProcess.cTitle
   * Insert this records into result
   INSERT INTO MgrHierarchy ;
      VALUES (m.cFirst, m.cLast, m.iLevel, ;
              m.cMgrFirst, m.cMgrLast, ;
              m.iCurrentID, m.iReportsTo, m.cTitle)

   * Grab the current record pointer
   nCurRecNo = RECNO("EmpsToProcess")

   INSERT INTO EmpsToProcess ;
      SELECT EmployeeID, FirstName, LastName, m.iLevel + 1, ;
             m.cFirst, m.cLast, ReportsTo, Title ;
         FROM Employees ;
         WHERE ReportsTo = m.iCurrentID

   * Restore record pointer
   GO m.nCurRecNo IN EmpsToProcess
ENDSCAN

SELECT MgrHierarchy
```

We need to further extend the graphing class to add colors; the new version is in gvNWEmpGraph3.PRG in the downloads for this session. First, we populate the color cursor. **Listing 28** shows the code added to LoadColorsAndShapes for that.

**Listing 28**. To use colors in the graphing class, we need to set up the color cursor.

```
INSERT INTO csrColors VALUES ("VP", "gold") && Vice President
INSERT INTO csrColors VALUES ("SM", "deepskyblue") && Sales Manager
INSERT INTO csrColors VALUES ("SR", "hotpink1") && Sales Representative
INSERT INTO csrColors VALUES ("SC", "sienna1") && Sales Coordinator
```

As the code shows, the cursor expects the a two-letter type, but the job title is much longer. The new ConvertTitleToAbbrev method, shown in **Listing 29**, accepts a job title and returns the appropriate abbreviation. It's written to address the possibility that we may have other kinds of vice presidents, managers, representatives or coordinators, and want to color them all the same. (Of course, two-letter codes may not be the right way for you to do this, so you could modify the cursor definition for your needs. It's likely in a production database that there'd be a table of job titles and the primary key or another identifier from that table could be used for the color look-up.)

**Listing 29**. The color cursor uses a two-letter code, so this method converts job titles to two-letter codes.

```
PROCEDURE ConvertTitleToAbbrev(m.tcTitle)
* Convert full job title to abbrev.
* Generalizing here a little from the actual NW list

LOCAL lcTitle, lcAbbrev

lcTitle = UPPER(m.tcTitle)
lcAbbrev = ''
```

```
DO CASE
CASE "VICE PRESIDENT" $ m.lcTitle
   lcAbbrev = "VP"
CASE "MANAGER" $ m.lcTitle
   lcAbbrev = "SM"
CASE "REPRESENTATIVE" $ m.lcTitle
   lcAbbrev = "SR"
CASE "COORDINATOR" $ m.lcTitle
   lcAbbrev = "SC"
ENDCASE

RETURN m.lcAbbrev
ENDPROC
```

The GetNodeColor method from gvGraph expects to receive the two-letter code. In this case, I've extended the method to accept the full job title, do the conversion and then call the original code. This version is show in **Listing 30**.

**Listing 30**. To make GetNodeColor work with the job title, we need to call ConvertTitleToAbbrev and then pass the result to the original GetNodeColor code.

```
*************************************************

PROCEDURE GetNodeColor(tcNodeType)
* Add translation of full title to shortcut

LOCAL lcTypeAbbrev, lcColor

lcTypeAbbrev = This.ConvertTitleToAbbrev(m.tcNodeType)
lcColor = DODEFAULT(m.lcTypeAbbrev)

RETURN m.lcColor
ENDPROC
```

The final change in this class is for GenerateGVNode to call GetNodeColor and use the result in the node definition. **Listing 31** shows the changed portion of the method; note the addition of both the color and style attributes to the line.

**Listing 31**. This section of the GenerateGVNode method retrieves the desired color and adds color information to the node definition.

```
lcNodeColor = This.GetNodeColor(cTitle)

lcNodeDef = "    " + m.lcNodeID + '[shape="' + m.lcNodeShape + ;
            '",color="' + m.lcNodeColor + '",style="filled' + ;
            '",label="' + m.lcNodeLabel + ;
            '",fontname="' + This.cFontName + '"];' + CHR(13) + CHR(10)
```

The code to produce the graph requires two changes from the previous version. First, as in the prior example, we need to change the class name and the name of the PRG containing the class. We also need to call NWEmpsWTitle, rather than NWEmps. The new code, shown in **Listing 32**, is included in the downloads for this session as RunGraphNWEmp3.PRG. The graph definition file is shown in **Listing 33**.

**Listing 32**, Run this code to produce the graph in **Figure 10**.

```
* Northwind Employee graph, version 3 (shapes and colors)

* First, collect the data
DO NWEmpsWTitle.PRG

* Now create the graph
LOCAL loGenerateGraph
loGenerateGraph = NEWOBJECT("GraphNWEmp3", "gvNWEmpGraph3.prg")

IF VARTYPE(loGenerateGraph) = 'O' AND NOT ISNULL(m.loGenerateGraph)
   loGenerateGraph.MakeGraph('mgrhierarchy','jpeg')
   IF EMPTY(loGenerateGraph.cErrorMessages)
      loGenerateGraph.ShowGraph()
   ELSE
      MESSAGEBOX(loGenerateGraph.cErrorMessages)
   ENDIF
ENDIF

RETURN
```

**Listing 33**. This graph definition file results in the graph shown in **Figure 10**.

```
digraph GVFile {
  graph[rankdir=TB];
  graph[label="Northwind Employees", labelloc=top, labeljust=left, fontsize=24];
      2[shape="box",color="gold",style="filled",label="Andrew 
Fuller",fontname="Arial"];
      1[shape="ellipse",color="hotpink1",style="filled",label="Nancy 
Davolio",fontname="Arial"];
      2 ->    1;
      3[shape="ellipse",color="hotpink1",style="filled",label="Janet 
Leverling",fontname="Arial"];
      2 ->    3;
      4[shape="ellipse",color="hotpink1",style="filled",label="Margaret 
Peacock",fontname="Arial"];
      2 ->    4;
      5[shape="ellipse",color="deepskyblue",style="filled",label="Steven 
Buchanan",fontname="Arial"];
      2 ->    5;
      8[shape="ellipse",color="sienna1",style="filled",label="Laura 
Callahan",fontname="Arial"];
      2 ->    8;
      6[shape="diamond",color="hotpink1",style="filled",label="Michael 
Suyama",fontname="Arial"];
      5 ->    6;
      7[shape="diamond",color="hotpink1",style="filled",label="Robert 
King",fontname="Arial"];
      5 ->    7;
      9[shape="diamond",color="hotpink1",style="filled",label="Anne 
Dodsworth",fontname="Arial"];
      5 ->    9;
}
```

As noted earlier, you can specify different colors for the shape's border and fill. **Figure 11** shows the result if you specify the fillcolor attribute rather than the color attribute. In this case, the border uses the default black, but the interior uses the specified color. To produce this result, change "color" to "fillcolor" in the third line of GenerateGVNode that's shown in **Listing 31**.

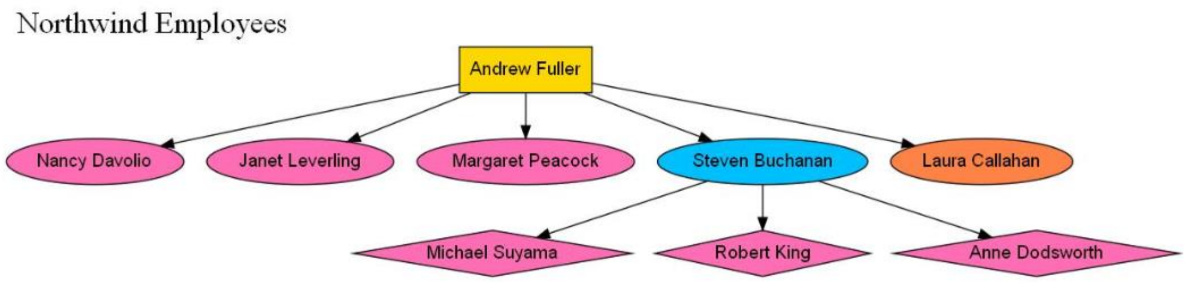


**Figure 11**. This shows what the graph would look like if the fillcolor attribute were used instead of the color attribute.

## Handling more complex graphs

Lots of things we want to graph are more complicated than an organizational chart. For example, some graphs involve nodes that both connect to and receive connections from multiple nodes. One such example is a database diagram, showing relationships between tables.

With the classes already shown, the hardest part of this problem is ordering the tables so that we only try to create edges for tables that already have nodes. It's easy to get the list of tables and the relationships in a database, but figuring out what order to put them in is complex.

As we'll see later in this section, to draw the graph, we need not just the cursor listing the tables in order, but one of the cursors that's created along the way. So, for this graph, it makes sense to collect the data right in the graphing class. To make that easier, this subclass of gvDotGraph, called GraphDB1 (included in the downloads for this session as gbDBGraph1.PRG), has a new property, cDatabase, and a method SetDatabase (shown in **Listing 34**) that accepts the name and path to the database and stores it in cDatabase.

**Listing 34**. This method accepts the name and path of a database, confirms the file exists, and saves it to the class's cDatabase property.

```
PROCEDURE SetDatabase(tcDatabase)
* Set the cDatabase property

LOCAL llSuccess

llSuccess = .T.

DO CASE
```

```
CASE EMPTY(m.tcDatabase)
   This.AddError("You must specify a database to graph.")
   llSuccess = .F.

CASE NOT FILE(FORCEEXT(tcDatabase, 'dbc'))
   This.AddError("You must pass a database (DBC) to SetDatabase.")
   llSuccess = .F.

OTHERWISE
   This.cDatabase = m.tcDatabase

ENDCASE
RETURN m.llSuccess
```

The code to make the list of tables goes in the GetGraphData method, shown in **Listing 35**. It does the job for any database that doesn't have any cyclical relationships (that is, where table a is related to table b, which is related to table c, which is related to table a).

**Listing 35**. This code builds a list of tables in a database and their relations in an order that lets us graph them.

```
* Construct data showing all tables and relations
* in specified database, in an order that lets us
* create the graph.
LPARAMETERS cDatabase && include path

* First grab relation information from the database
OPEN DATABASE (m.cDataBase)
LOCAL aRelns[1]
ADBOBJECTS(aRelns,'relation')
CREATE CURSOR csrRelns ;
   (cChild C(128), cParent C(128), cChildTag C(10), cParentTag C(10))
INSERT INTO csrRelns FROM ARRAY aRelns
CLOSE DATABASES


* Next build a list of tables, showing how many relations
* each is involved in, both as parent and child
SELECT cParent, COUNT(*) AS nParentCount ;
   FROM csrRelns ;
   GROUP BY 1 ;
   INTO CURSOR csrParentCount

SELECT cChild, COUNT(*) as nChildCount ;
   FROM csrRelns ;
   GROUP BY 1 ;
   INTO CURSOR csrChildCount

SELECT objectname as cTable, ;
       NVL(nParentCount,0) as nParentCount, ;
       NVL(nChildCount,0) as nChildCount ;
   FROM FORCEEXT(m.cDatabase, 'dbc');
     LEFT JOIN csrParentCount ;
       ON UPPER(objectname) = UPPER(cParent) ;
     LEFT JOIN csrChildCount ;
       ON UPPER(objectname) = UPPER(cChild) ;
   WHERE objecttype = 'Table' ;
   INTO CURSOR csrTables READWRITE

* Start building the list of tables with those
* that don't appear as children or are only
* children in a self-join
SELECT UPPER(cTable) as cTable ;
   FROM csrTables ;
   WHERE nChildCount = 0 ;
UNION ALL ;
SELECT UPPER(cTable) AS cTable ;
   FROM csrTables ;
     JOIN (SELECT * FROM csrRelns ;
              WHERE cChild = cParent) csrSelfJoin ;
        ON UPPER(cTable) = cParent ;
   WHERE nChildCount = 1 ;
INTO CURSOR csrTableOrder READWRITE

LOCAL lDone
lDone = .F.

* Use the list so far to pull in other tables
* for which we now have all parents, until
* all tables are in the list or we can't
* get our hands on the others because of complex
* relationships
DO WHILE NOT m.lDone
   IF RECCOUNT("csrTableOrder") = RECCOUNT("csrTables")
      lDone = .T.

   ELSE
      SELECT DISTINCT cChild AS cTable ;
         FROM csrRelns ;
           JOIN csrTableOrder ;
             ON csrTableOrder.cTable = csrRelns.cParent ;
         WHERE cChild NOT IN (SELECT * FROM csrTableOrder csrTO) ;
           AND NOT EXISTS (SELECT cParent FROM csrRelns csrR ;
                             WHERE csrR.cChild = csrRelns.cChild ;
                               AND csrR.cParent NOT IN (;
                                   SELECT cTable FROM csrTableOrder csrTO2)) ;
         INTO CURSOR csrAddChildren

      IF _TALLY = 0
         lDone = .T.
      ENDIF

      SELECT csrTableOrder
      APPEND FROM DBF("csrAddChildren")

   ENDIF
ENDDO

RETURN
```

The code has four sections. The first two are fairly simple. First, we use ADBObjects() to get a list of relations in the database and put it into a cursor. Next, we use that information and the database itself to build a list of the tables in the database with a count of how many times that table appears as a parent in a relation and how times it appears as a child.

Step 3 is to put into our result cursor a list of those tables we can start with when drawing nodes. That's any table that appears only as a parent, and any table that appears as a child only in a relationship with itself (like the Employee table we worked with earlier in this paper).

The last step is the hard part. We need to put the rest of the tables into the cursor in the right order. The idea is that a table can only be added to the cursor if every table that's a parent to that table has already been added and, of course, we only want to add each table once.  The query inside the loop finds such tables.

The DO WHILE loop continues until either all tables have been added or the query doesn't find anything else to add.

Once we have the data, it doesn't take much more than what we've already done to produce a graph of it. **Figure 12** shows a graph of the tables in the Northwind database.

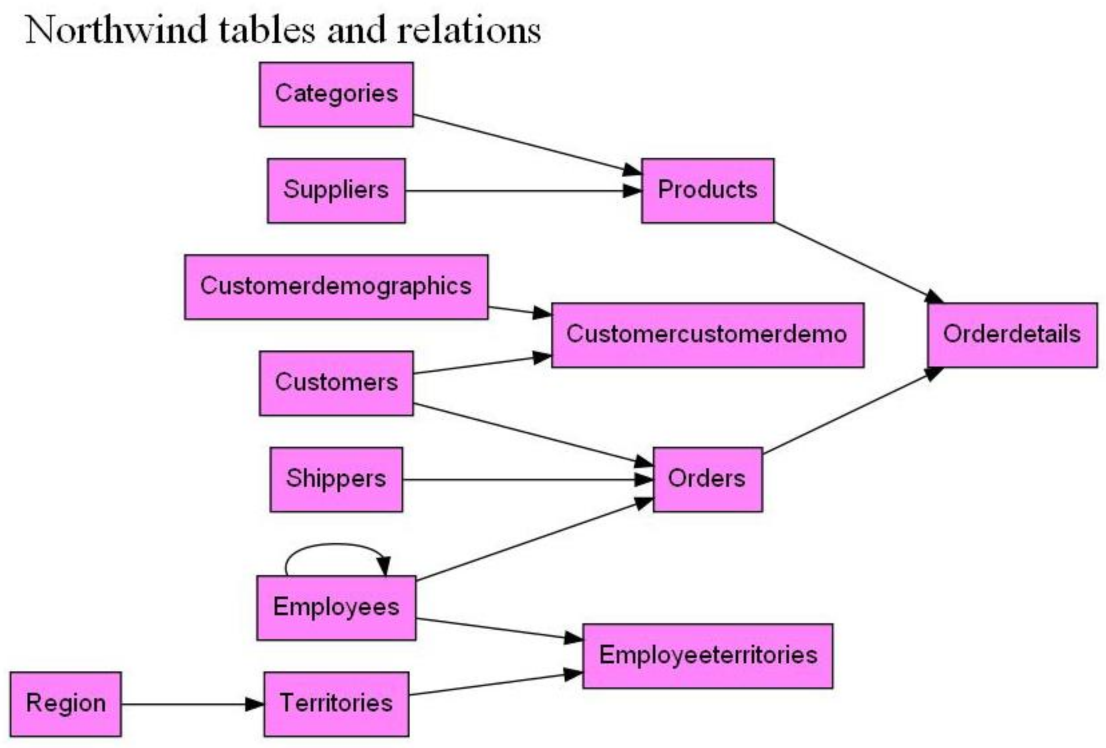


**Figure 12.** This graph shows the tables in the Northwind database and the relations between them.

This graph is designed to go left-to-right rather than top-to-bottom, so the parameter to SetGraphOrientation in the Setup method is 'LR'. In addition, the Setup method uses the cDatabase property to construct the title to pass to SetGraphTitle.

The table name is used for both the node ID and the node label. **Listing 36** shows the GetNodeID and GetNodeLabel methods.

**Listing 36**. The table name is used for both the node ID and the node label, though it's set to proper case for the label.

```
PROCEDURE GetNodeID(tcTable)

RETURN ALLTRIM(m.tcTable)
ENDPROC

*************************************************
PROCEDURE GetNodeLabel
* Use the table name as the label
LOCAL lcLabel

lcLabel = PROPER(ALLTRIM(cTable))

RETURN m.lcLabel
ENDPROC
```

Generating each node in the GenerateGVNode method (**Listing 37**) isn't much different than previous examples. Here, every node uses the same shape and fill color, so those values are hard-coded. Note the use of the fillcolor attribute, rather than color; as you can see in the figure, the border of each node is black.

**Listing 37**. As before, the GenerateGVNode method creates the definition line for a single node. Here, the shape and color are hard-coded.

```
PROCEDURE GenerateGVNode
* Generate GV line to create a single node,
* based on the current line of the cursor

LOCAL lcNodeDef, lcNodeID, lcNodeLabel

* Make sure we're in the right workarea
LOCAL lnOldSelect

lnOldSelect = SELECT()
SELECT (This.cDataCursor)

lcNodeID = This.GetNodeID(cTable)
lcNodeLabel = This.GetNodeLabel()

lcNodeDef = "   " + m.lcNodeID + '[label="' + m.lcNodeLabel + ;
            '",shape="box",style="filled",fillcolor="orchid1"' + ;
            'fontname="' + This.cFontName + '"];' + CHR(13) + CHR(10)
This.WriteGraphLine(m.lcNodeDef)
```

```
SELECT (m.lnOldSelect)

RETURN m.lcNodeID

RETURN
ENDPROC
```

The most interesting work for this class happens in GenerateGVNodeEdges (**Listing 38**). It uses the node ID parameter to retrieve the list of relations where this table is a child, that is, a list of parent tables for this table, and then loops through the resulting cursor, generating an edge for each parent.

**Listing 38.** To show the relations between tables, we use the cursor of relations that was created while building the data.

```
PROCEDURE GenerateGVNodeEdges(tcNodeID)
* Generate the edges for the specified node

LOCAL lcEdgeLine

LOCAL lnOldSelect
lnOldSelect = SELECT()

* Retrieve cases where this is a child node from the relations cursor
SELECT cParent ;
   FROM csrRelns ;
   WHERE cChild == m.tcNodeID ;
   INTO CURSOR csrCurParents

LOCAL lcParentID

SCAN
   lcParentID = This.GetNodeID(csrCurParents.cParent)
   lcEdgeLine = "   " + m.lcParentID + " -> " + m.tcNodeID + ";" + CHR(13) + CHR(10)
This.WriteGraphLine(m.lcEdgeLine)
ENDSCAN

USE IN (SELECT("csrCurParents"))


SELECT (m.lnOldSelect)

RETURN
ENDPROC
```

The last step is code to instantiate and call the class. That's a short program (RunDGGraph1.PRG in the downloads for this session), shown in **Listing 39**. It accepts the database, including path, as a parameter.

**Listing 39.** Pass the database, including path, to this code to generate the graph of tables.

```
* Database graph, version 1 (basic)
LPARAMETERS tcDatabase
```

```
* Now create the graph
LOCAL loGenerateGraph
loGenerateGraph = NEWOBJECT("GraphDB1", "gvDBGraph1.prg")

IF VARTYPE(loGenerateGraph) = 'O' AND NOT ISNULL(m.loGenerateGraph)
   IF loGenerateGraph.SetDatabase(m.tcDatabase)
      loGenerateGraph.MakeGraph(.f.,'jpeg')
   ENDIF
   IF EMPTY(loGenerateGraph.cErrorMessages)
      loGenerateGraph.ShowGraph()
   ELSE
      MESSAGEBOX(loGenerateGraph.cErrorMessages)
   ENDIF
ENDIF

RETURN
```

The graph definition for the Northwind database is shown in **Listing 40**.

**Listing 40**. This graph definition produces the graph shown in **Figure 12**.

```
digraph GVFile {
  graph[rankdir=LR];
  graph[label="Northwind tables and relations", labelloc=top, labeljust=left,
fontsize=24];

CATEGORIES[label="Categories",shape="box",style="filled",fillcolor="orchid1"fontname=
"Arial"];

CUSTOMERS[label="Customers",shape="box",style="filled",fillcolor="orchid1"fontname="A
rial"];

SHIPPERS[label="Shippers",shape="box",style="filled",fillcolor="orchid1"fontname="Ari
al"];

SUPPLIERS[label="Suppliers",shape="box",style="filled",fillcolor="orchid1"fontname="A
rial"];

CUSTOMERDEMOGRAPHICS[label="Customerdemographics",shape="box",style="filled",fillcolo
r="orchid1"fontname="Arial"];

REGION[label="Region",shape="box",style="filled",fillcolor="orchid1"fontname="Arial"]
;

EMPLOYEES[label="Employees",shape="box",style="filled",fillcolor="orchid1"fontname="A
rial"];
   EMPLOYEES -> EMPLOYEES;

CUSTOMERCUSTOMERDEMO[label="Customercustomerdemo",shape="box",style="filled",fillcolo
r="orchid1"fontname="Arial"];
   CUSTOMERDEMOGRAPHICS -> CUSTOMERCUSTOMERDEMO;
   CUSTOMERS -> CUSTOMERCUSTOMERDEMO;
ORDERS[label="Orders",shape="box",style="filled",fillcolor="orchid1"fontname="Arial"]
;
   CUSTOMERS -> ORDERS;
   SHIPPERS -> ORDERS;
   EMPLOYEES -> ORDERS;

PRODUCTS[label="Products",shape="box",style="filled",fillcolor="orchid1"fontname="Ari
al"];
   CATEGORIES -> PRODUCTS;
   SUPPLIERS -> PRODUCTS;

TERRITORIES[label="Territories",shape="box",style="filled",fillcolor="orchid1"fontnam
e="Arial"];
   REGION -> TERRITORIES;

EMPLOYEETERRITORIES[label="Employeeterritories",shape="box",style="filled",fillcolor=
"orchid1"fontname="Arial"];
   TERRITORIES -> EMPLOYEETERRITORIES;
   EMPLOYEES -> EMPLOYEETERRITORIES;

ORDERDETAILS[label="Orderdetails",shape="box",style="filled",fillcolor="orchid1"fontn
ame="Arial"];
   PRODUCTS -> ORDERDETAILS;
   ORDERS -> ORDERDETAILS;
}
```

## Making edges more informative

In the previous example, while the edges show us the connections, they don't provide much other information. Like nodes, edges can take attributes. We can use those attributes to improve the graph.

### Adding edge labels

One obvious improvement is to show the names of the tags that the relationship between tables is based on, as in **Figure 13**.

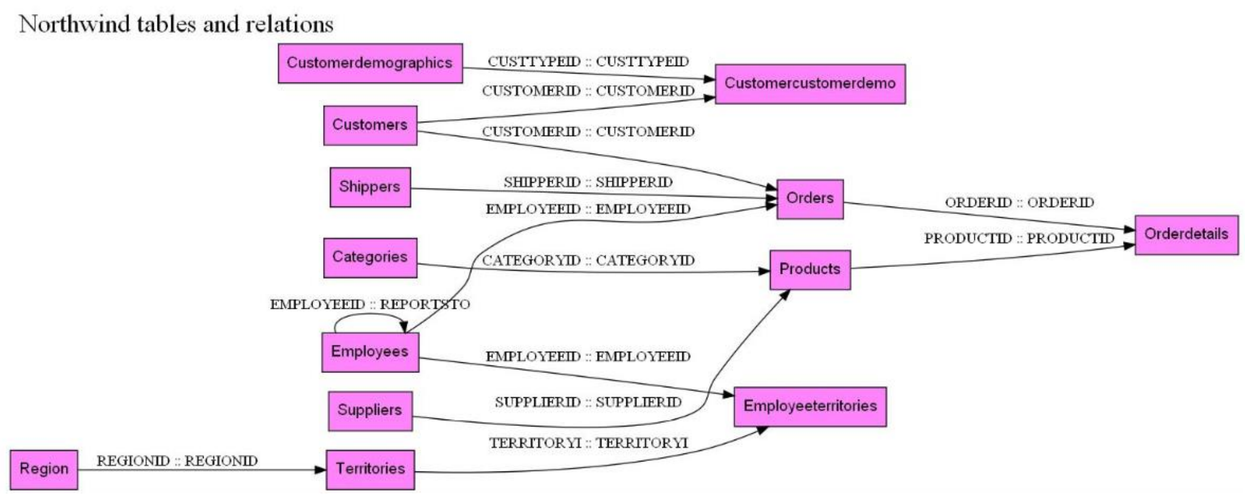


**Figure 13**. This version of the database diagram includes information about the tags used to create relations.

It only takes a little more code to add this information. Like nodes, edges can have a label attribute. In the class GraphDB2 (included in the downloads for this session in gvDBGraph2.PRG), the GenerateGVNodeEdges method has a little additional code to include the label. The key part of the method is shown in **Listing 41**. It adds the cChildTag and cParentTag fields to the query and sets up the desired separator (“ :: “). Inside the loop, the label for that edge is constructed and the label attribute is added to the definition of the edge.

**Listing 41**. Like nodes, edges can have labels.

```
SELECT cParent, cChildTag, cParentTag ;
   FROM csrRelns ;
   WHERE cChild == m.tcNodeID ;
   INTO CURSOR csrCurParents

LOCAL lcParentID, lcSeparator
lcSeparator = " :: "

SCAN
   lcParentID = This.GetNodeID(csrCurParents.cParent)
   lcLabel = ALLTRIM(cParentTag) + m.lcSeparator + ALLTRIM(cChildTag)
   lcEdgeLine = "    " + m.lcParentID + " -> " + m.tcNodeID + ;
                '[label="' + m.lcLabel + '"]' + ;
                ";" + CHR(13) + CHR(10)
   This.WriteGraphLine(m.lcEdgeLine)
ENDSCAN
```

**Listing 42** shows a few edge definitions from the graph definition file.

**Listing 42**. Edges accept attributes, just as graphs and nodes do.

```
   CUSTOMERS -> ORDERS[label="CUSTOMERID :: CUSTOMERID"];
   SHIPPERS -> ORDERS[label="SHIPPERID :: SHIPPERID"];
   EMPLOYEES -> ORDERS[label="EMPLOYEEID :: EMPLOYEEID"];
```

## Changing arrowheads (and tails)

By default, in a directed graph, edges have an arrowhead at one end, and it uses a traditional triangular arrowhead. But you can change the style of the arrowhead, and you can also put a shape at the other end of the edge. GraphViz calls that an arrowtail. **Figure 14** shows the database diagram using both arrowheads and arrowtails, with each end indicating whether that side of the relationship is "one" (the vertical bar) or "many" (the triangle).

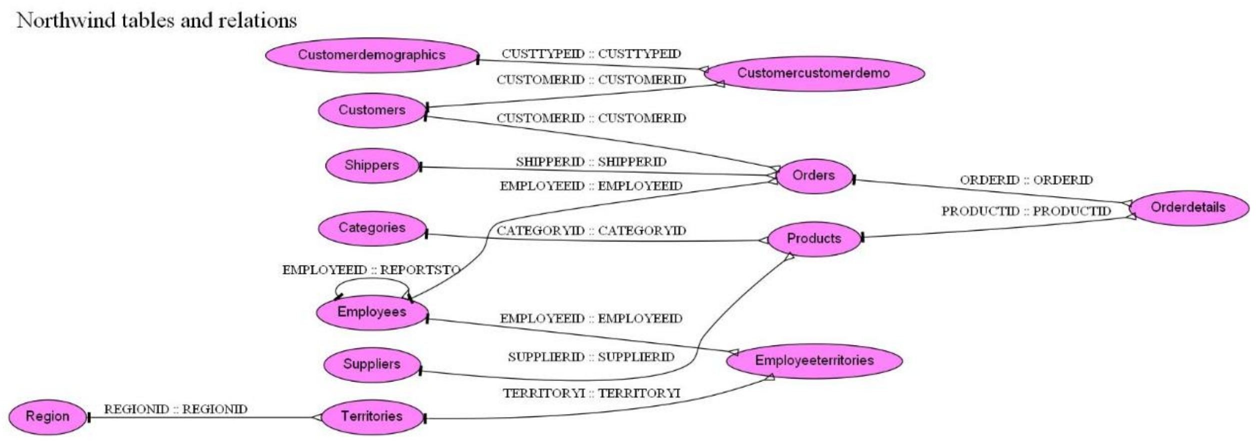


**Figure 14.** In this version of the database diagram, the edges indicate the type of relation: one-to-one, one-to-many, or many-to-many.

Three edge attributes are involved in providing this information. The dir attribute specifies which ends of an edge should show arrows. Specify "both" to allow both arrowheads and arrowtails, "forward" for arrowheads only, or "back" for arrowtails only.

The arrowhead and arrowtail attributes let you indicate which shapes to use for a given edge. A set of common options are provided and documented at http://graphviz.org/doc/info/attrs.html#k:arrowType. You can also define your own; that's documented at http://graphviz.org/doc/info/arrows.html.

To add appropriate shapes, we need additional code in the GenerateGVNodeEdges that checks whether the tags involved in the relation are primary or candidate keys (in which case, that side of the relation is "one"). The code that generates the edge than uses that information. **Listing 43** shows this complete version of the method in the class GraphDB3, which is included in the downloads for this session in gvDBGraph3.prg. Because the "tee" shape used for a "one" side of a relationship doesn't show up well against a rectangle, the GenerateGVNode method has been changed to use the "oval" shape instead of "box".

**Listing 43.** This version of the method figures out whether each end of the relationship is one or many and sets an appropriate arrow shape.

```
PROCEDURE GenerateGVNodeEdges(tcNodeID)
* Generate the edges for the specified node

LOCAL lcEdgeLine
LOCAL lnOldSelect
lnOldSelect = SELECT()

* Retrieve cases where this is a child node from the relations cursor
SELECT cParent, cChildTag, cParentTag ;
   FROM csrRelns ;
   WHERE cChild == m.tcNodeID ;
   INTO CURSOR csrCurParents

LOCAL lcParentID, lcSeparator
lcSeparator = " :: "

LOCAL llParentSingle, llChildSingle, lnTagNo
LOCAL lcArrowHead, lcArrowTail
LOCAL lcParent

OPEN DATABASE (This.cDatabase)
SCAN
   lcParentID = This.GetNodeID(csrCurParents.cParent)
   lcLabel = ALLTRIM(cParentTag) + m.lcSeparator + ALLTRIM(cChildTag)

   * Figure out whether one or many at each end

   USE (tcNodeID) IN 0
   lnTagNo = TAGNO(ALLTRIM(csrCurParents.cChildTag), m.tcNodeID, m.tcNodeID)
   llChildSingle = PRIMARY(m.lnTagNo, m.tcNodeID) OR CANDIDATE(m.lnTagNo, m.tcNodeID)
   USE IN SELECT(m.tcNodeID)

   lcParent = ALLTRIM(csrCurParents.cParent)
   USE (m.lcParent) IN 0
   lnTagNo = TAGNO(ALLTRIM(csrCurParents.cParentTag), m.lcParent, m.lcParent)
   llParentSingle = PRIMARY(m.lnTagNo, m.lcParent) OR ;
                    CANDIDATE(m.lnTagNo, m.lcParent)
   USE IN SELECT(m.lcParent)

   IF m.llParentSingle
      lcArrowTail = "tee"
   ELSE
      lcArrowTail = "invempty"
   ENDIF

   IF m.llChildSingle
      lcArrowHead = "tee"
   ELSE
      lcArrowHead = "invempty"
   ENDIF

   lcEdgeLine = "   " + m.lcParentID + " -> " + m.tcNodeID + ;
                '[label="' + m.lcLabel + '",' + ;
                'dir="both",arrowhead="' + m.lcArrowHead + ;
                '",arrowtail="' + m.lcArrowTail +'"' + ;
                "];" + CHR(13) + CHR(10)
   This.WriteGraphLine(m.lcEdgeLine)
ENDSCAN
```

```
CLOSE DATABASES

USE IN (SELECT("csrCurParents"))

SELECT (m.lnOldSelect)

RETURN
ENDPROC
```

**Listing 44** shows the graph definition file for this version.

**Listing 44**. This graph definition generates the graph in **Figure 14**, which includes arrowheads and arrowtails that indicate the type of relation.

```
digraph GVFile {
  graph[rankdir=LR];
  graph[label="Northwind tables and relations", labelloc=top, labeljust=left, 
fontsize=24];

CATEGORIES[label="Categories",shape="oval",style="filled",fillcolor="orchid1"fontname
="Arial"];

CUSTOMERS[label="Customers",shape="oval",style="filled",fillcolor="orchid1"fontname="
Arial"];

SHIPPERS[label="Shippers",shape="oval",style="filled",fillcolor="orchid1"fontname="Ar
ial"];

SUPPLIERS[label="Suppliers",shape="oval",style="filled",fillcolor="orchid1"fontname="
Arial"];

CUSTOMERDEMOGRAPHICS[label="Customerdemographics",shape="oval",style="filled",fillcol
or="orchid1"fontname="Arial"];

REGION[label="Region",shape="oval",style="filled",fillcolor="orchid1"fontname="Arial"
];

EMPLOYEES[label="Employees",shape="oval",style="filled",fillcolor="orchid1"fontname="
Arial"];
   EMPLOYEES -> EMPLOYEES[label="EMPLOYEEID :: 
REPORTSTO",dir="both",arrowhead="invempty",arrowtail="tee"];

CUSTOMERCUSTOMERDEMO[label="Customercustomerdemo",shape="oval",style="filled",fillcol
or="orchid1"fontname="Arial"];
   CUSTOMERDEMOGRAPHICS -> CUSTOMERCUSTOMERDEMO[label="CUSTTYPEID :: 
CUSTTYPEID",dir="both",arrowhead="invempty",arrowtail="tee"];
   CUSTOMERS -> CUSTOMERCUSTOMERDEMO[label="CUSTOMERID :: 
CUSTOMERID",dir="both",arrowhead="invempty",arrowtail="tee"];

ORDERS[label="Orders",shape="oval",style="filled",fillcolor="orchid1"fontname="Arial"
];
   CUSTOMERS -> ORDERS[label="CUSTOMERID :: 
CUSTOMERID",dir="both",arrowhead="invempty",arrowtail="tee"];
```

```
   SHIPPERS -> ORDERS[label="SHIPPERID ::
SHIPPERID",dir="both",arrowhead="invempty",arrowtail="tee"];
   EMPLOYEES -> ORDERS[label="EMPLOYEEID ::
EMPLOYEEID",dir="both",arrowhead="invempty",arrowtail="tee"];

PRODUCTS[label="Products",shape="oval",style="filled",fillcolor="orchid1"fontname="Ar
ial"];
   CATEGORIES -> PRODUCTS[label="CATEGORYID ::
CATEGORYID",dir="both",arrowhead="invempty",arrowtail="tee"];
   SUPPLIERS -> PRODUCTS[label="SUPPLIERID ::
SUPPLIERID",dir="both",arrowhead="invempty",arrowtail="tee"];

TERRITORIES[label="Territories",shape="oval",style="filled",fillcolor="orchid1"fontna
me="Arial"];
   REGION -> TERRITORIES[label="REGIONID ::
REGIONID",dir="both",arrowhead="invempty",arrowtail="tee"];

EMPLOYEETERRITORIES[label="Employeeterritories",shape="oval",style="filled",fillcolor
="orchid1"fontname="Arial"];
   TERRITORIES -> EMPLOYEETERRITORIES[label="TERRITORYI ::
TERRITORYI",dir="both",arrowhead="invempty",arrowtail="tee"];
   EMPLOYEES -> EMPLOYEETERRITORIES[label="EMPLOYEEID ::
EMPLOYEEID",dir="both",arrowhead="invempty",arrowtail="tee"];

ORDERDETAILS[label="Orderdetails",shape="oval",style="filled",fillcolor="orchid1"font
name="Arial"];
   PRODUCTS -> ORDERDETAILS[label="PRODUCTID ::
PRODUCTID",dir="both",arrowhead="invempty",arrowtail="tee"];
   ORDERS -> ORDERDETAILS[label="ORDERID ::
ORDERID",dir="both",arrowhead="invempty",arrowtail="tee"];
}
```

## Enhancing the graph

The only graph-level attributes we've specified so far are rankdir, which controls the orientation of the graph, and a few related to labeling the graph. But of course, there are other things we can do to make the graph more attractive or informative.

### Controlling the label

A bunch of attributes control labels. Those you're most likely to want to adjust are listed in **Table 2**. Many of these attributes can be applied to nodes and edges, as well as to the graph itself.

**Table 2**. Use these attributes to control the appearance of labels.

| Attribute | Values | Meaning |
|---|---|---|
| label | <any string> | The text for the specified label |
| labeljust | l, c, r | Indicates whether the label is left, center, or right justified |
| labelloc | t, c, b | Indicates whether the label goes at the top, center, or bottom. For the whole graph, the only choices are t (top) and b (bottom) |
| fontname | <an available font> | Specifies the font used for text |

| | | |
|---|---|---|
| fontcolor | <a named color> | Specifies the color used for text |
| fontsize | <a number> | Specifies the size of text |

**Figure 15** shows another version of the database diagram. All labels in this graph use Arial, and the graph label is centered and shown in blue.

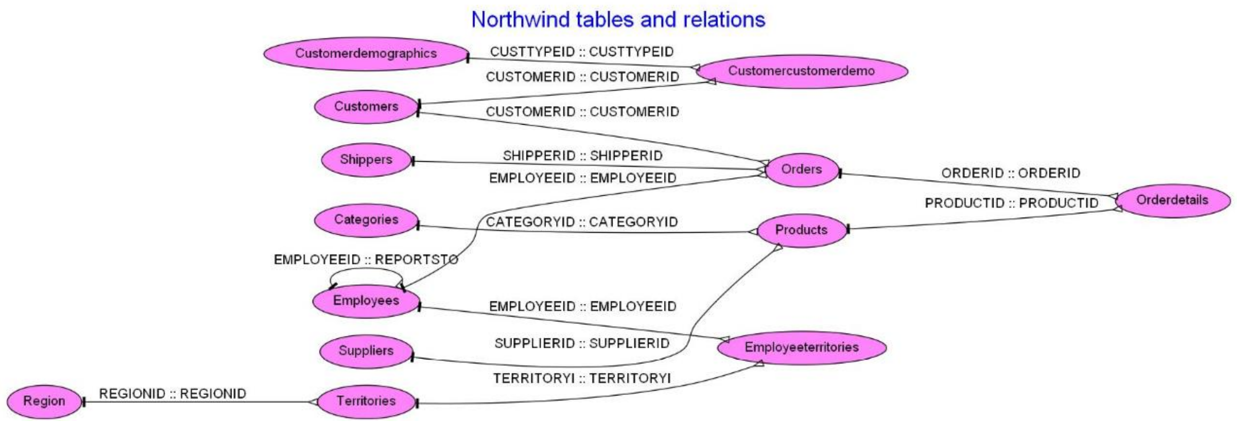


**Figure 15**. In this version of the database diagram, the graph's label is centered and uses blue text, and all labels (graph, node, and edges) use the same font.

This version uses the class GraphDB4 (contained in gvDBGraph4.PRG in the downloads for this session) and is created by running RunDBGraph4.PRG. GenerateGVNodeEdges has been changed to include the fontname attribute for edges. In addition, the GenerateGVHeader method has been overridden with the version shown in **Listing 45**.

**Listing 45**. This method sets the labeljust attribute of the graph to center, and uses blue Arial for the label.

```
PROCEDURE GenerateGVHeader
* Put the header info into the GV file

LOCAL lcLine

lcLine = 'digraph GVFile {' + CHR(13) + CHR(10)
* First line starts a new file
This.WriteGraphLine(m.lcLine, .T.)

lcLine = '  graph[rankdir=' + This.cGraphOrientation + '];' + CHR(13) + CHR(10)
This.WriteGraphLine(m.lcLine)

IF NOT EMPTY(This.cGraphTitle)
   lcLine = '  graph[label="' + This.cGraphTitle + ;
            '", labelloc=top, labeljust=center, ' + ;
            'fontname="Arial",fontcolor="blue",fontsize=24];' + CHR(13) + CHR(10)
   This.WriteGraphLine(m.lcLine)
ENDIF

RETURN
ENDPROC
```

## Other graph attributes

You can specify the background color for a graph using the bgcolor attribute. It uses the same list of colors as the color, fillcolor, and fontcolor attributes described earlier in this document.

If you're dealing with a lot of data, you might need to stretch your graph over multiple pages. The rotate attribute lets you rotate your graph 90 degrees.

**Figure 16** shows one final version of the database diagram rotated 90 degrees and with the background color set to ivory. To produce this version, the GenerateGVHeader method was modified (in class GraphDB5 in gvDBGraph5.PRG in the downloads for this session) to add the two additional attributes; the updated portion of the code is shown in **Listing 46**. The graph definition file is shown in **Listing 47**.

**Listing 46**. To rotate the graph and set the background color, attributes are added to the graph section.

```
lcLine = '  graph[label="' + This.cGraphTitle + ;
         '", labelloc=top, labeljust=center, ' + ;
         'fontname="Arial",fontcolor="blue",fontsize=24,' + ;
         'bgcolor="ivory",rotate=90' + ;
         '];' + CHR(13) + CHR(10)
```

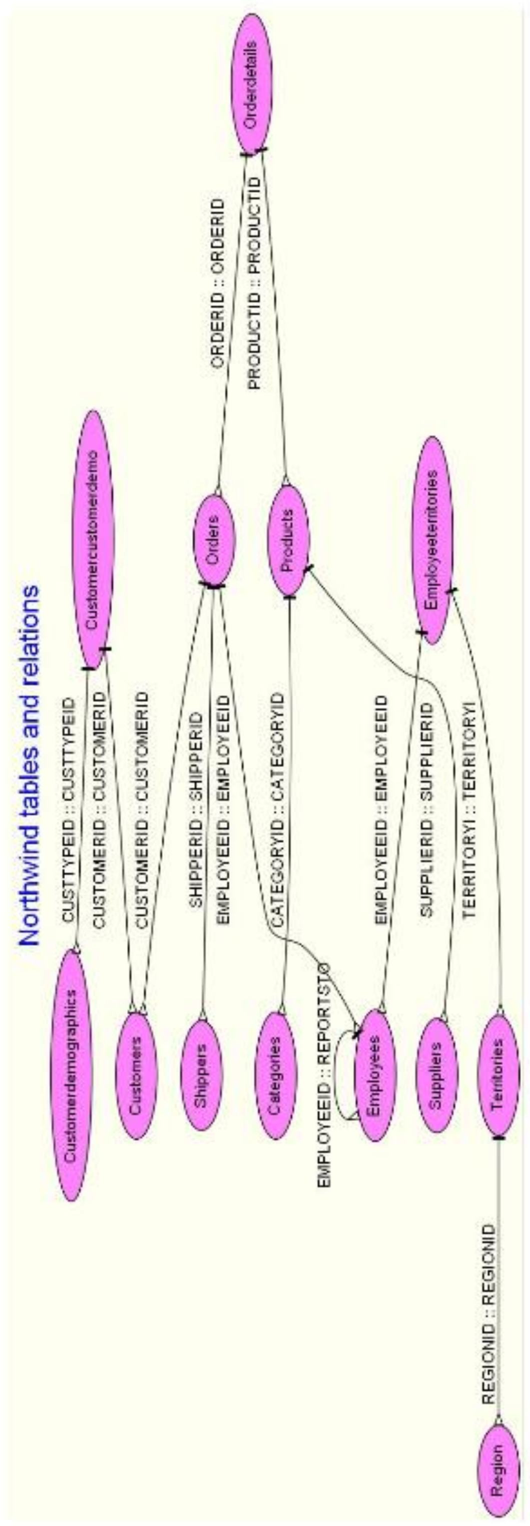


**Figure 16.** You can rotate a graph to make it easier to print, as well as set a background color.

If I planned to use any of these additional attributes often, I'd enhance the class gvGraph to make it easier to set them rather than hard-coding the changes.

**Listing 47**. This version of the graph definition file produces the graph shown in **Figure 16**.

```
digraph GVFile {
  graph[rankdir=LR];
  graph[label="Northwind tables and relations", labelloc=top, labeljust=center,
fontname="Arial",fontcolor="blue",fontsize=24,bgcolor="ivory",rotate=90];

CATEGORIES[label="Categories",shape="oval",style="filled",fillcolor="orchid1"fontname
="Arial"];

CUSTOMERS[label="Customers",shape="oval",style="filled",fillcolor="orchid1"fontname="
Arial"];

SHIPPERS[label="Shippers",shape="oval",style="filled",fillcolor="orchid1"fontname="Ar
ial"];

SUPPLIERS[label="Suppliers",shape="oval",style="filled",fillcolor="orchid1"fontname="
Arial"];

CUSTOMERDEMOGRAPHICS[label="Customerdemographics",shape="oval",style="filled",fillcol
or="orchid1"fontname="Arial"];

REGION[label="Region",shape="oval",style="filled",fillcolor="orchid1"fontname="Arial"
];

EMPLOYEES[label="Employees",shape="oval",style="filled",fillcolor="orchid1"fontname="
Arial"];
   EMPLOYEES -> EMPLOYEES[label="EMPLOYEEID ::
REPORTSTO",dir="both",arrowhead="tee",arrowtail="invempty",fontname="Arial"];

CUSTOMERCUSTOMERDEMO[label="Customercustomerdemo",shape="oval",style="filled",fillcol
or="orchid1"fontname="Arial"];
   CUSTOMERDEMOGRAPHICS -> CUSTOMERCUSTOMERDEMO[label="CUSTTYPEID ::
CUSTTYPEID",dir="both",arrowhead="tee",arrowtail="invempty",fontname="Arial"];
   CUSTOMERS -> CUSTOMERCUSTOMERDEMO[label="CUSTOMERID ::
CUSTOMERID",dir="both",arrowhead="tee",arrowtail="invempty",fontname="Arial"];

ORDERS[label="Orders",shape="oval",style="filled",fillcolor="orchid1"fontname="Arial"
];
   CUSTOMERS -> ORDERS[label="CUSTOMERID ::
CUSTOMERID",dir="both",arrowhead="tee",arrowtail="invempty",fontname="Arial"];
   SHIPPERS -> ORDERS[label="SHIPPERID ::
SHIPPERID",dir="both",arrowhead="tee",arrowtail="invempty",fontname="Arial"];
   EMPLOYEES -> ORDERS[label="EMPLOYEEID ::
EMPLOYEEID",dir="both",arrowhead="tee",arrowtail="invempty",fontname="Arial"];

PRODUCTS[label="Products",shape="oval",style="filled",fillcolor="orchid1"fontname="Ar
ial"];
   CATEGORIES -> PRODUCTS[label="CATEGORYID ::
CATEGORYID",dir="both",arrowhead="tee",arrowtail="invempty",fontname="Arial"];
   SUPPLIERS -> PRODUCTS[label="SUPPLIERID ::
SUPPLIERID",dir="both",arrowhead="tee",arrowtail="invempty",fontname="Arial"];

TERRITORIES[label="Territories",shape="oval",style="filled",fillcolor="orchid1"fontna
me="Arial"];
```

```
   REGION -> TERRITORIES[label="REGIONID ::
REGIONID",dir="both",arrowhead="tee",arrowtail="invempty",fontname="Arial"];

EMPLOYEETERRITORIES[label="Employeeterritories",shape="oval",style="filled",fillcolor
="orchid1"fontname="Arial"];
   TERRITORIES -> EMPLOYEETERRITORIES[label="TERRITORYI ::
TERRITORYI",dir="both",arrowhead="tee",arrowtail="invempty",fontname="Arial"];
   EMPLOYEES -> EMPLOYEETERRITORIES[label="EMPLOYEEID ::
EMPLOYEEID",dir="both",arrowhead="tee",arrowtail="invempty",fontname="Arial"];

ORDERDETAILS[label="Orderdetails",shape="oval",style="filled",fillcolor="orchid1"font
name="Arial"];
   PRODUCTS -> ORDERDETAILS[label="PRODUCTID ::
PRODUCTID",dir="both",arrowhead="tee",arrowtail="invempty",fontname="Arial"];
   ORDERS -> ORDERDETAILS[label="ORDERID ::
ORDERID",dir="both",arrowhead="tee",arrowtail="invempty",fontname="Arial"];
}
```

## What else can GraphViz do?

The examples in this paper show only a subset of the capabilities provided by GraphViz. Among the most important items not covered here is the ability to create subgraphs (including a special set of subgraphs called clusters) that let you specify sections within graphs. One of the uses of clusters is to create legends for graphs.

Because they're easiest to show in a paper, all the examples here produce JPEG output, but as noted earlier, GraphViz supports a wide range of output capabilities. Depending on your goal, some of them may be more appropriate.

In this paper, we've looked only at the dot engine, but GraphViz includes several other engines. My sense is that dot is likely to be the right choice for most graphs that come out of database applications, but if you're not getting what you want, look at the other engines.