Creating A File Packing List

by Paul Warren

Back in Issue 23 I wrote about `TEventList`, fully intending to be back with more tips for developing component suites. Alas, time slipped by without any new ideas and I finally decided to release the components as freeware so I could move on to other projects.

And just when I thought everything was done, *BOOM*, the email poured in. 'I tried to compile your components and...' 'When I tried to compile your components...' 'After trying to compile I get error message...' I guess you get the picture.

Suffice it to say I learned another difference between component suites and single components: they are more difficult to package and release. The problem was I forgot to include two files I had squirreled away in lesser used directories. Before you issue me my programmer's dunce cap let me say in my defence that I did try the package on another computer. Unfortunately there were old versions of the files present on the path so I didn't get a compile error during testing.

As the redness in my face subsided I promised myself I would find a way to prevent this from ever happening again. What I really needed was a utility to create a packing list for any future product releases.

Accept a pascal source file name from the user
Find the file on a path list
Add the file name to an Outline at the appropriate level
Add the file name to an exclude list to prevent circular references
Obtain the uses clause from the file
Parse the first unit, not in the exclude list, from the uses clause
Repeat until all units have been visited

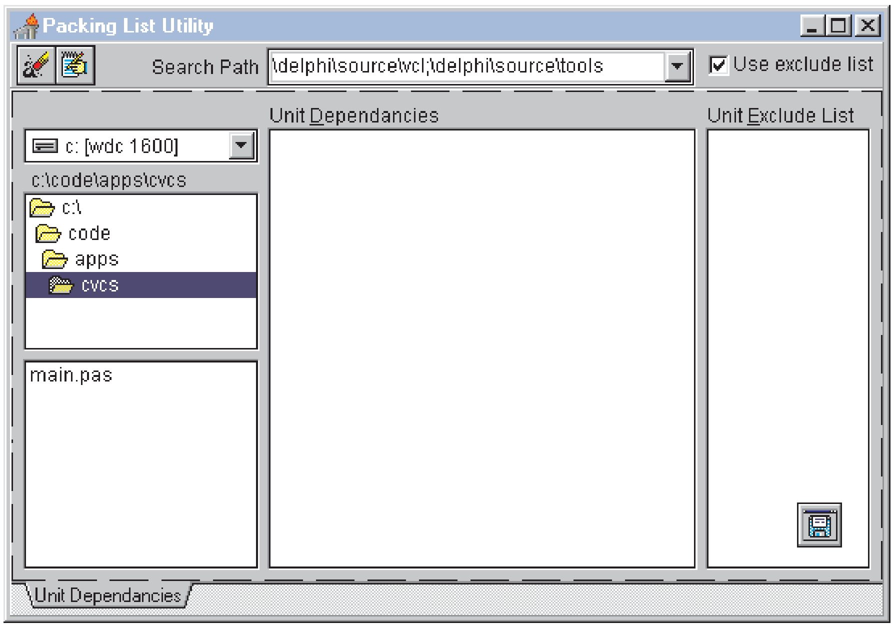
➤ *Table 1: File packing list requirements*

Creating A Packing List

To create a packing list for a component suite a utility would need to extract the `uses` clause of a given source file, parse out each unit in turn, write the units to some data structure and repeat this process until all units have been found. See Table 1 for a requirements list. If you think this sounds like 'walking' a directory tree then you are right.

There are a couple of other requirements as well. Such a utility would need to exclude the Borland supplied units and other third party units if desired. It should also be able to supply useful output, printed or otherwise. Finally, it should only report found units once. The prototype interface is shown in Figure 1.

➤ *Figure 1*



The Recursive Engine

Central to the utility is the recursive method

```
BuildList(FName: string;
  Level: integer)
```

When called with a unit filename and `Level` of 1, `Buildlist` constructs an Outline of unit dependencies. `BuildList` is shown in Listing 1.

As you can see `BuildList` first locates the unit with `FindUnit` and then writes the unit filename to `Outline1` and sets `Outline1.Items[Idx]` to `Level`. It then obtains the `uses` clause from `GetUsesClause`. The next step is to parse the first unit from the `uses` clause leaving any other units in the variable `Remainder`. If any of these steps fail, for example if there is no `uses` clause, we have an empty string and the procedure will end at the line if `NextString` is not empty.

If `NextString` is not blank the procedure calls itself with `NextString` as `FName` and `Inc(Level)` as `Level`. When the end of each branch is reached the procedure exits, decrements the level and parses the next unit from `Remainder` at the previous level. In this way every unit in the tree is visited.

Obtaining The Uses Clause

By far the most difficult routine is the `GetUsesClause` function.

Originally I tried paging the source file into a buffer and using my REGEXP16.DLL search engine to find the expression uses. Then a series of Move operations left a string containing the uses clause.

The problem with this is that REGEXP16.DLL is completely unaware of Pascal syntax so it finds uses anywhere. If uses is found in a comment ahead of the true uses clause I got garbage. The returned string also contains CR/LF pairs and whitespace that have to be removed.

While I did get the function to work this way it was unreliable and even took out the operating system occasionally.

The solution for making GetUsesClause work was presented by Marco Cantù in Issue 23. In his article on converting Pascal source to HTML he showed how to use the TParser class. TParser is syntax aware (more or less). With just a little work TParser can cleanly extract the uses clause from any source file. Using TParser has the added bonus of eliminating CR/LFs and whitespace and makes it possible to extract the uses clause from both the interface and implementation sections. Listing 2 shows the complete GetUsesClause function. As it's quite complex I'll go through it in some detail.

```
{ BuildList - recursive routine to “walk” the units tree }
procedure TCVCSMain.BuildList(FName: string; var Level: integer);
var
  NextFile, Remaining: string;
  Idx: integer;
begin
  FindUnit(FName); { find the file on the path }
  Idx := Outline1.Add(Outline1.SelectedItem, FName);  { add node for file }
  Outline1.Items[Idx].Level := Level;
  { add file to exclude list to avoid infinite
    recursion from circular unit references }
  ExcludeList.Add(FName);
  Remaining := GetUsesClause(FName); { get the uses clause from FName }
  NextFile := Parse(Remaining);      { parse the units clause }
  while NextFile <> '' do begin
    { if NextFile is not empty... }
    Inc(Level);   { Inc tree level }
    BuildList(NextFile, Level);     { recurse with first dependant file }
    NextFile := Parse(Remaining);   { find next dependant file }
    Dec(Level);   { Dec tree level }
  end;
end;
```

➤ *Listing 1*

After setting some control variables, GetUsesClause attempts to create a TFileStream inside a try...except block. If TFileStream.Create fails the function returns an empty string, allowing the recursive BuildList to exit gracefully and continue processing (don't forget to turn off *break on exception* if running under the IDE, otherwise the program stops every time a file is not found).

If TFileStream.Create is successful we create an instance of TParser with the TFileStream as a parameter. Inside a

```
while Token <> toEOF do ...
```

loop we repeatedly call NextToken and manipulate the control variables BeginCopy, FoundUses and IsComment according to the value of Token. Note that when we find the first semi-colon after the toSymbol implementation we exit. This makes it unnecessary to process the entire file. We can get away with this because there can't be anything but compiler directives or comments between implementation and uses.

There is only one small problem. TParser doesn't like single quotes. For my own purposes this doesn't matter. I'm happy to replace the few occurrences of single quotes with two quotes, especially since TParser kindly reports the offending line number and the last unit processed is the unit TParser failed to read. If you don't like this behavior you will need to modify TParser or, as Marco Cantù suggested,

➤ *Listing 2*

```
{ GetUsesClause - routine to extract uses clause from unit }
function TCVCSMain.GetUsesClause(FName: string): string;
var
  AStream: TFileStream;
  Parser: TParser;
  BeginCopy,
  FoundUses,
  IsComment: boolean;
  S, OutStr: string;
begin
  { initiallize variables }
  Result := '';
  BeginCopy := false;
  FoundUses := false;
  IsComment := false;
  try
    { open FileStream(FName) }
    AStream := TFileStream.Create(FName, fmOpenRead);
    Parser := TParser.Create(AStream);  {create unit parser}
    try
      with Parser do
        while Token <> toEOF do begin
          S := TokenString;
          case Token of
            toSymbol :
              begin
                if (TokenString = 'implementation') and not
                  IsComment then
                  FoundUses := true;
                if (TokenString = 'uses') and not IsComment
                  then begin
                  BeginCopy := true;
                  S := '';
                end;
              end;
            ';' :
              begin
                if FoundUses then
                  Exit;
                if BeginCopy then begin
                  AppendStr(Result, ',');
                  BeginCopy := false;
                end;
              end;
            '{' :
              begin
                S := '';
                IsComment := true;
              end;
            '}' :
              begin
                S := '';
                IsComment := false;
              end;
          end;
          if BeginCopy and not IsComment then
            AppendStr(Result, S);
          NextToken;
        end;
    finally
      Parser.Free;
      AStream.Free;
    end;
  except
    { on file open error return empty string }
    on EFOpenError do Result := '';
  end;
end;
```

write a better implementation from scratch.

Parsing The Uses Clause

After extracting the uses clause from a file we need to parse it into its constituent units. Listing 3 shows the `Parse` function. It takes a comma delimited string and separates the first item, returning the parsed string with the remainder of the original delimited string.

The function does two other things as well. First, a call to `IsValidIdent` checks that the parsed string is a valid unit name and appends the .pas extension if it is. Second, the entire parsing code is inside a `repeat...until` loop that checks the result against an exclude list. This is where we can eliminate units that belong to the RTL or third parties. `Parse` will return an empty string if any of the tests fail, again so `BuildList` can continue processing.

```
{ Parse - routine to parse the uses clause }
function TCVCSMain.Parse(var ParseStr: string): string;
var Len: integer;
begin
  Result := '';
  if Length(ParseStr) > 0 then begin
    { if there is something to parse... }
    repeat
      if Pos(',', ParseStr) <> 0 then begin
        { if there is a comma copy up to it }
        Len := Pos(',', ParseStr);
        Result := System.Copy(ParseStr, 1, Len-1);
      end else begin
        { else copy all remaining string }
        Len := Length(ParseStr);
        Result := System.Copy(ParseStr, 1, Len);
      end;
      System.Delete(ParseStr, 1, Len);      { delete what we copied }
      { if we have a valid unit name... }
      if IsValidIdent(Result) then
        Result := Result+'.pas' { add .pas extension }
      else
        Result := ''; { else return empty string }
      { ...until there is a unit NOT in the exclude list }
    until (ExcludeList.IndexOf(Result) < 0);
  end;
end;
```

➤ *Listing 3*

```
procedure TCVCSMain.FindUnit(var FName: string);
var
  FN, TempStr: string;
begin
  FN := FName; { set FN equal to FName }
  { perform the search }
  TempStr := FileSearch(FN, Edit1.Text);
  { if successful change FName }
  if TempStr <> '' then FName := ExpandFileName(TempStr);
end;
```

➤ *Listing 4*

Searching For The Units

If a unit is not in the current directory then we have to search for it. Here I decided to use a path string the same way the IDE does. The main reason for using a path is to take advantage of the built-in `FileSearch` function which takes a path as the second parameter.

As you can see from Listing 4 the `FindUnit` procedure searches the user defined path and changes `FName` to the fully qualified file name if `FileSearch` is successful. If unsuccessful `FName` is unchanged. This way any file with a full path was found, any file without a path has not been found: useful information for the packing list.

The Exclude List

The exclude list serves two purposes. As already mentioned, it allows the user to exclude units which are not of interest. Secondly, and more importantly, the exclude list prevents the infinite recursion that would surely occur sooner or later from a circular unit reference.

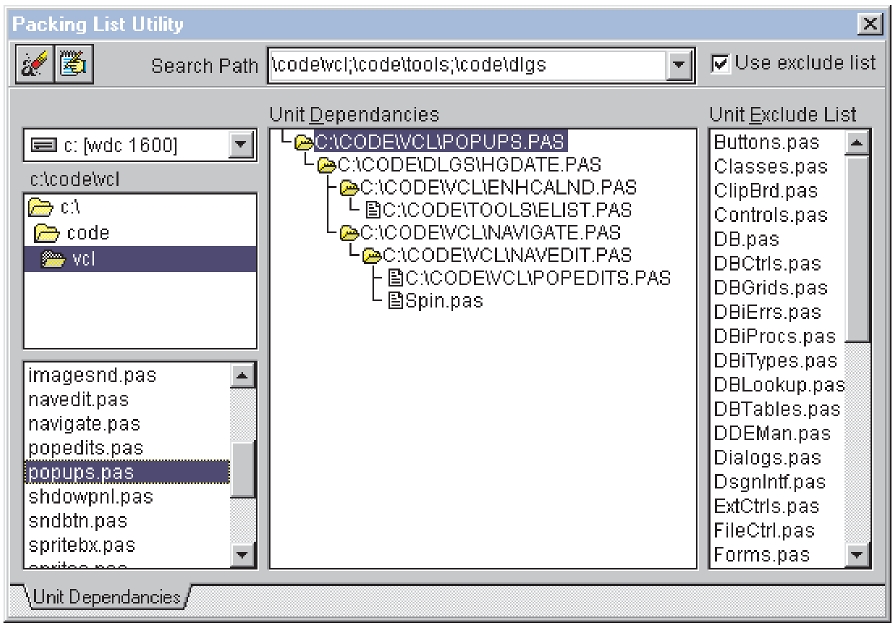
Remember, the compiler will generate a circular unit reference error if one unit references another which in turn references itself. That is unless the circular reference is in the `implementation` section. Since `GetUsesClause` combines *both* clauses, a circular reference would be a disaster.

By adding each processed unit in turn to the exclude list the `Parse` function will never return the same unit twice. Not only does this eliminate the problem of circular references but it solves the design problem of only listing a given unit once. You can see the utility in action in Figure 2.

Output

While the utility provides a nice outline view of unit dependencies, to be truly useful some other form of output is needed. In keeping

➤ *Figure 2*



Search Path: \code\vcl;\code\tools;\code\dlgs

[✓] Use exclude list

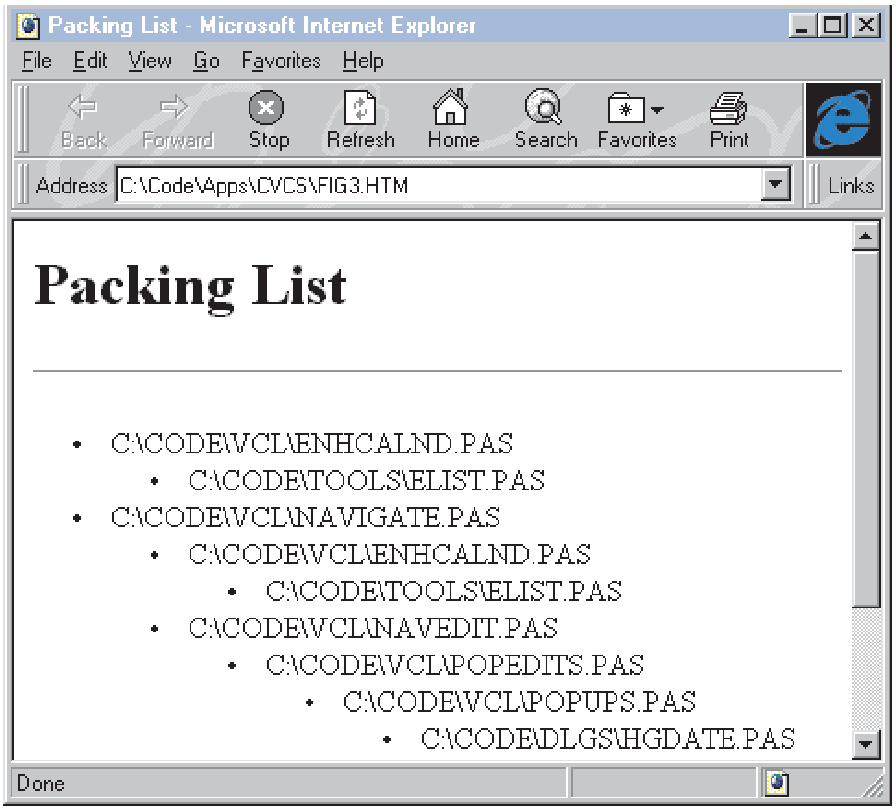
```
imagesnd.pas
navedit.pas
navigate.pas
popedits.pas
popups.pas
shdowpnl.pas
sndbtn.pas
spritebx.pas
```

```
└C:\CODE\VCL\POPUPS.PAS
  └C:\CODE\DLGS\HGDATE.PAS
    ├C:\CODE\VCL\ENHCALND.PAS
    │ └C:\CODE\TOOLS\ELIST.PAS
    └C:\CODE\VCL\NAVIGATE.PAS
      └C:\CODE\VCL\NAVEDIT.PAS
        ├C:\CODE\VCL\POPEDITS.PAS
        └Spin.pas
```

```
Buttons.pas
Classes.pas
ClipBrd.pas
Controls.pas
DB.pas
DBCtrls.pas
DBGrids.pas
DBiErrs.pas
DBiProcs.pas
DBiTypes.pas
DBLookup.pas
DBTables.pas
DDEMan.pas
Dialogs.pas
DsgnIntf.pas
ExtCtrls.pas
FileCtrl.pas
Forms.pas
```

```
{ OutputHtml - routine to write ouptu in Html format }
procedure TCVCSMain.OutputHtml(FName: string);
var
  F: TextFile;
  i, j, CurLev: integer;
begin
  CurLev := 0;
  AssignFile(F, FName);
  Rewrite(F);
  { write Html header }
  writeln(F, '<HTML>');
  writeln(F, '<HEAD><TITLE>Packing List</TITLE>');
  writeln(F, '</HEAD>');
  writeln(F, '<BODY BGCOLOR="#FFFFFF" TEXT="#000000" ‘+
    ‘LINK="#0000FF" VLINK="#00009B" ALINK="#DA0000">');
  writeln(F, '<H1>'+Format('Packing List for %s',
    [ChangeFileExt(FileListBox1.Items[
    FileListBox1.ItemIndex], '.zip')])+'</H1>');
  writeln(F, '<HR>');
  try
    { iterate through the outline }
    for i := 0 to Outline1.Lines.Count-1 do begin
      { if level goes up... }
      if Outline1.Items[i+1].Level > CurLev then begin
        write(F, '<UL>'); { increase indent }
        Inc(CurLev);      { increase CurLev }
      end;
      { if level goes down... }
      if Outline1.Items[i+1].Level < CurLev then
        { for CurLev down to the new level }
        for j := CurLev downto Outline1.Items[i+1].Level+1
          do begin
          write(F, '</UL>'); { close list level }
          Dec(CurLev);  { decrease CurLev }
        end;
      write(F, #13#10);
      { write out the actual text }
      write(F, '<LI>'+Outline1.Items[i+1].Text);
    end;
    for j := CurLev downto 0 do
      write(F, '</UL>'); { close all list levels }
    { Html footer }
    write(F, '<HR>'#13#10'Generated by CVCS from');
    writeln(F, ' HomeGrown Software, by Paul Warren.');
    writeln(F, '</BODY></HTML>');
  finally
    CloseFile(F);
  end;
end;
```

➤ *Listing 5*


Packing List - Microsoft Internet Explorer
File Edit View Go Favorites Help
Back Forward Stop Refresh Home Search Favorites Print
Address C:\Code\Apps\CVCS\FIG3.HTM
Links
Done

Packing List

- C:\CODE\VCL\ENHCALND.PAS
  - C:\CODE\TOOLS\ELIST.PAS
- C:\CODE\VCL\NAVIGATE.PAS
  - C:\CODE\VCL\ENHCALND.PAS
    - C:\CODE\TOOLS\ELIST.PAS
  - C:\CODE\VCL\NAVEDIT.PAS
    - C:\CODE\VCL\POPEDITS.PAS
      - C:\CODE\VCL\POPUPS.PAS
        - C:\CODE\DLGS\HGDATE.PAS

➤ *Figure 3*

with the spirit of the web I decided to provide HTML output. This way I can easily provide packing lists on my web page or use my browser to print the formatted list. Figure 3 shows the ouput for part of my calendar component suite. Note the missing file hgDate.pas shows up clearly. Oh how I wish I had created this utility before...

Listing 5 is the `OutputHtml` procedure. After writing a file header we simply iterate through the `Outline1.Items` list and write the unit names to a list. The level of the `Outline1.Items` controls the indentation of the list. As the level increases the code adds a `<UL>` tag and increments `CurLev`. Since the level can only increase by 1 this is sufficient.

The level can decrease by any amount however, hence the `for CurLev downto ...Level` statement. We have to add one `</UL>` tag for each step down. After the whole list has been output we add a `</UL>` tag for each level down to 0, completing nested lists. All that is left is to write out a footer.

Conclusion

Whether you are releasing a simple freeware component or a full commercial component suite, your hard work will be for nought and your reputation will suffer if your package isn't complete. One way to protect your investment is to generate a packing list prior to packaging and distribution.

While there are any number of improvements you could make, I think this utility should help you avoid the embarrassment I suffered with my botched release of Calendar Suite v1.0. My only regret is not having developed this utility first.

A final word: this project will compile under Delphi 1, just turn off stack checking and increase the stack size.

Paul Warren runs HomeGrown Software Development in Langley, British Columbia, Canada and can be contacted by email at hg_soft@haven.uniserve.com