Extending the Lazarus IDE: The code tools

Michaël Van Canneyt

December 8, 2005

Abstract

In a previous article, the lazarus source editor was explored. In this article, the matter is pushed to the next level by introducing the code tools. The code tools provide methods to handle and manipulate pascal-specific source code. At the same time, it is shown how one can easily modify the source editor window by adding a toolbar to it.

1 Introduction

In a previous article, the source editor was explored. The source editor provides a simple interface to the Lazarus IDE source editor: it offers the possibility to manipulate the sources at the text level: lines, selection, text position. No understanding of pascal sources exists at this level.

The handling of the pascal code in the code editor happens at the level of the code tools. The code tools are a separate package, available outside the IDE. They can operate on text independent of the lazarus IDE.

The code tools are a huge package. It would lead far beyond the scope of this article to explain all the features in the code tools. The interested reader can browse the sources in the directory

```
components/codetools
```

of the lazarus source tree. There are too many files to discuss here. The main files of interest are:

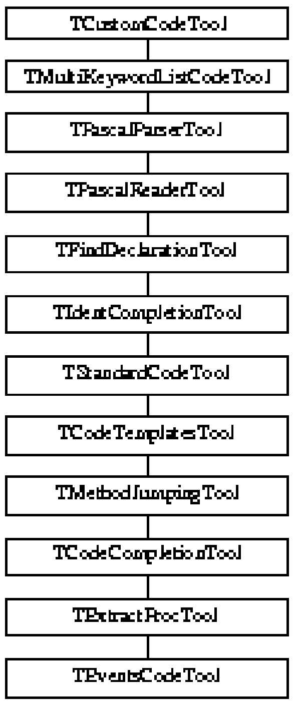
CustomCodeTool introduces the `TCustomCodeTool` class, which corresponds roughly to a syntactic parser for source files. It has a notion of various source files, and a notion of 'dirty' files, i.e. files which need rescanning. It doesn't do the actual scanning itself.

CodeToolManager this unit contains the `TCodeToolManager` which manages the various code tools. It also contains the `TCodeTool` class, which is an alias for the actual code tool used in the IDE.

CodeTree contains the `TCodeTree` and `TCodeTreeNode` classes. As the Pascal code is parsed and analysed, a tree of nodes is constructed. Each node corresponds with an construct of the pascal language: a statement, expression, uses clause etc.

CodeAtom Defines the smallest syntactical element of the pascal code:`TCodeAtum`. This can be an identifier, keyword or comment. It defines the position of this element in the pascal code. A `TCodeTreeNode` usually contains one or more `TCodeAtoms`

Figure 1: Code tools hierarchy



CodeBuffer introduces the `TCodeBuffer`, which roughly corresponds to a source file. A unit will typically be represented by one or more `TCodeBuffer` instances: one instance for the main file, other instances for the various include files.

Each of these files is needed when working with the code tools, as various classes contained in these units are needed.

2 The Code Tool classes

The `TCustomCodeTool` class is the ancestor of a large line of code tool classes. The complete tree is presented in figure 1 on page 2. Each class in the tree introduces a new level of understanding and manipulation of pascal code, and adds more complex handling of the code.

The separation in different classes is done to clearly separate the various levels of functionality, and keeps the code clean by implementing each class in a separate unit: Each class resides in a unit that has the same name as the class (dropping the `T` in front of the name).

The code tool class currently used by the editor interface is of type `TEventsCodeTool`. An alias exists in the codetoolmanager unit: `TCodeTool`. This alias may change when the the code tools are upgraded, therefore will always point to the actual class used.

Each of the classes implements some functionality:

TMultiKeyWordListCodeTool Implements the handling of a keyword list.

TPascalParserTool Implements elementary pascal parsing.

TPascalReaderTool Extends the `TPascalParsingTool` with methods for accessing the positions in the sourcecode. Can be used to position the editor on various parts of the source.

TFindDeclarationTool enhances the `TPascalReaderTool` class with the ability to find the source position or code tree node of an identifier's declaration.

TIdentCompletionTool enhances the `TFindDeclarationTool` with the ability to create lists of valid identifiers at a specific code position.

TStandardCodeTool enhances `TIdentCompletionTool` with many standard code editing functions, e.g. search and replace, finding lists of identifiers or types of objects.

TCodeTemplatesTool enhances `TStandardCodeTool` with the ability to insert code templates.

TMethodJumpingCodeTool enhances `TCodeTemplatesTool` with functions to jump between a method or procedure definition and its implementation.

TCodeCompletionCodeTool enhances `TMethodJumpingCodeTool` with code completion functions: completing properties, methods, unimplemented functions, but also local variables.

TExtractProcTool enhances `TCodeCompletionCodeTool` with functionality to extract statements from procedures and to move them to new procedures, local procedures or methods.

TEventsCodeTool enhances `TExtractProcTool` with functions to work with published methods in the source. It can gather a list of compatible methods, test if method exists, jump to the method body, create a method, complete all missing published variables and events from a root component.

This is a long list of classes. It should be obvious from the description that each of these implements a part of the IDE's editor functionality. In practice, only `TPascalReadTool` or higher will be used to browse or manipulate pascal source.

The IDE itself uses the functions in these classes to implement all it's behaviour and to drive the syntax editor - which is totally separate from the code tools. There are many hundreds of functions, most of them used by the IDE.

Obviously, it's impossible to treat all these functions here. The class tree listing above should give a rough idea of where to look for specific functionality in the code tools. In the next section, it will be explained how to obtain a reference to the code tools, and how to initialize them so they can be used.

3 Implementing fast jump functionality

In this section, a method will be developed to implement a button which allows to quickly jump to the interface or implementation keywords in a unit, and to jump to the uses clause of the interface or implementation.

This is useful when one needs to insert a new unit in the uses clause of the unit. If the unit is very large, it takes a lot of time to scroll to the uses clause. The jump function will make this action faster.

To do this, a new package is made. In this package, a `TJumpHandler` component is made:

```
TJumpHandler = Class(TComponent)
Private
  Function JumpToNode (Tool : TCodeTool;
                       Node : TCodeTreeNode) : Boolean;
Public
  Procedure DoJump(Sender : TObject);
end;
```

The reason for making this a `TComponent` is twofold:

1. A method is needed to be able to create a `TNotifyEvent` handler.
2. It is a `TComponent` descendent, so it can have the application object as the owner. The application will then destroy the `TJumpHandler` when the application object is freed.

The `DoJump` method does most of the work:

```
procedure TJumpHandler.DoJump(Sender: TObject);

var
  SrcEditor: TSourceEditorInterface;
  CodeBuffer: TCodeBuffer;
  CurCodeTool: TCustomCodeTool;
  Node: TCodeTreeNode;
  Tool: TCodeTool;
  Ok: Boolean;
  T : Integer;

begin
  T:=-1;
  If (Sender<>Nil) and (Sender is TComponent) then
    T:=TComponent(Sender).Tag;
  If Not (T in [1..4]) then
    exit;
  if not LazarusIDE.BeginCodeTools then
    exit;
  SrcEditor:=SourceEditorWindow.ActiveEditor;
  if Not Assigned(SrcEditor) then
    exit;
```

It starts with some security checks. The tag of the sender component is used to encode where to jump to:

1. The `interface` keyword.
2. The interface `uses` clause.
3. The `implementation` keyword.
4. The implementation `uses` clause.

After checking whether the code tools have been initialized it obtains a reference to the current source editor.

Then the actual work starts by obtaining a reference to the codebuffer (which represents the source file in the editor for the code tools) and initializing the codetools for this buffer:

```
  CodeBuffer:=SrcEditor.CodeToolsBuffer as TCodeBuffer;
  Ok:=false;
  try
    if CodeToolBoss.InitCurCodeTool(CodeBuffer) then
      begin
      CurCodeTool:=CodeToolBoss.CurCodeTool;
      if CurCodeTool is TCodeTool then
        begin
        Tool:=TCodeTool(CurCodeTool);
```

In the last lines, a `TCodeTool` instance is obtained for the current codebuffer. Now everything is in place to do the actual jump:

```
        Case T of
          1 : Node:=Tool.FindInterfaceNode;
          2 : Node:=Tool.FindMainUsesSection;
          3 : Node:=Tool.FindImplementationNode;
          4 : Node:=Tool.FindImplementationUsesSection;
        end;
        if (Node<>Nil) then
          OK:=JumpToNode(Tool,Node)
        else
          ShowMessage(Format(SErrCouldNotFind,[JumpNames[T]]));
        end;
      end;
  except
    on E: Exception do
      CodeToolBoss.HandleException(E);
  end;
  if not Ok then
    LazarusIDE.DoJumpToCodeToolBossError;
end;
```

To jump to a specific syntactical element in the source, the node for this element is determined, and stored in `Node`. The node is obtained through one of the methods of `TPascalReaderTool`. The `MainUsesSection` function returns the uses section of the unit or of the program, if the current source file contains a program.

If a node is found, then the `JumpToNode` function is invoked, which will be explained below. If no node is found, a message is displayed. `JumpNames` is an array with the names of the jump locations, which will be used once more, later on.

If somewhere along the way, the codetools encounter an error and raise an exception, it is caught, and the `DoJumpToCodeToolBossError` method of the lazarus IDE is invoked: this will analyse the codetool error, and attempt to jump to the location in the sources that triggered the error.

If everything goes well, the `JumpToNode` method is invoked to do the actual jump:

```
Function JumpToNode (Tool : TCodeTool;
                     Node : TCodeTreeNode) : Boolean;

Var
  NewTopLine: Integer;
  NewCodePos: TCodeXYPosition;
  SrcEditor: TSourceEditorInterface;
```

```
begin
  NewTopLine:=0;
  NewCodePos:=CleanCodeXYPosition;
  Result:=Tool.CleanPosToCaretAndTopLine(Node.StartPos,
                                        NewCodePos,NewTopLine);
  if Result then
    Result:=LazarusIDE.DoOpenFileAndJumpToPos(NewCodePos.Code.Filename,
                      Point(NewCodePos.X,NewCodePos.Y),NewTopLine,-1,
                      [ofRegularFile,ofUseCache])=mrOk;
  If Result then
    begin
    SrcEditor:=SourceEditorWindow.ActiveEditor;
    if Assigned(SrcEditor) then
      SrcEditor.EditorControl.SetFocus;
    end;
end;
```

The `SourceEditorWindow` contains the `TForm` descendent with the Source Editor Window in it. It exposes a single function, namely `ActiveEditor`, of type `TSourceEditorInterface`, which was discussed in an earlier article.

The `TCodeXYPosition` record is defined in the `CodeAtom` unit, and represents a logical position in a source file:

```
TCodeXYPosition = record
  X, Y: integer;
  Code: TCodeBuffer;
end;
```

The meaning of `X,Y` should be obvious, and `TCodeBuffer` corresponds to a file as seen by the code tools. The `CleanCodeXYPosition` function returns an initialized `TCodeXYPosition` record, with a non-existing (empty) position.

This record is then filled with the position of the node to which a jump should be performed with the `CleanPosToCaretAndTopLine` method of `TCustomCodeTool`. This method also stores a top line for the editor window. After this is done, the IDE interface (discussed in an earlier article) is asked to open the file and jump to the indicated position, with the `DoOpenFileAndJumpToPos` method. The `ofRegularFile` option tells the IDE to open the file as a regular source file. The `ofUseCache` option instructs the IDE to use it's internal file cache: if the file was already opened, it is not opened again.

If all this was succesful, the focus is set to the current editor, and all is done.

4 Adding a toolbar to the editor window

Now that a jump function is available, it should be made available in the editor. In previous articles, it was shown how to assign functions to keystrokes, or how to add functions to the local IDE menu or main menu. In this article, the function will be added to a toolbar in the IDE source editor window. This window does not have a toolbar by default, but the Lazarus IDE interface exposes the IDE Editor form instance. This makes it possible to extend the form with whatever functionality one wishes.

To do this, the `Register` procedure of the tbeditor unit is implemented as follows:

```
Procedure Register;
```

```
begin
  If (SourceEditorWindow<>Nil) then
    InitEditorToolBar;
end;
```

The `SourceEditorWindow` variable is of type `TSourceEditorWindowInterface` and is defined in the srceditorintf unit. The window is created before the packages are registered.

The InitEditorToolbar does the actual work:

```
Procedure InitEditorToolbar;

Var
  W  : TForm;
  TB : TToolbar;
  BI : TImageList;
  B  : TToolButton;
  PM : TPopupMenu;

begin
  JumpHandler:=TJumpHandler.Create(Application);
  W:=SourceEditorWindow;
  CreateEditorToolBar(W,TB,BI);
  TB.BeginUpdate;
  Try
    B:=TToolbutton.Create(W);
    B.Parent:=TB;
    B.Caption:='Jump';
    B.ImageIndex:=BI.AddFromLazarusResource('tbpos');
    B.Style:=tbsDropDown;
    PM:=TPopupMenu.Create(W);
    B.DropdownMenu:=PM;
    PM.Items.Add(CreateJumpItem(1,W));
    PM.Items.Add(CreateJumpItem(2,W));
    PM.Items.Add(CreateJumpItem(3,W));
    PM.Items.Add(CreateJumpItem(4,W));
  Finally
    TB.EndUpdate;
  end;
```

First, the jumphandler which was defined in the previous section is created. Then, a toolbar and imagelist are created in the `CreateEditorToolbar` procedure. The `BeginUpdate` and `EndUpdate` are placed there to avoid flicker when more than one button would be placed on the toolbar. A `TToolbutton` instance is placed on the toolbar, and a dropdown-menu is created and filled with items. Each item is created in a call to `CreateJumpItem`. This function accepts as arguments the location to jump to (the tag) and the owner for the menu item. The button image is included as a lazarus resource, named `tbpos`.

The `CreateEditorToolbar` function is very simple:

```
Function CreateEditorToolbar(W : TForm;
                             Var TB : TToolbar;
                             Var BI : TImageList);
```

```
begin
  BI:=TImageList.Create(W);
  TB:=TToolbar.Create(W);
  TB.Parent:=W;
  TB.Height:=26;
  TB.Align:=alTop;
  TB.Flat:=True;
  TB.Images:=BI;
end;
```

and the `CreateJumpItem` is just as simple:

```
Function CreateJumpItem(ATag : Integer; O : TComponent) : TMenuItem;

begin
  Result:=TMenuItem.Create(O);
  Result.Tag:=ATag;
  Result.OnClick:=@JumpHandler.DoJump;
  Result.Caption:=JumpNames[ATag];
end;
```

Note that the `JumpHandler.DoJump` is used as the `OnClick` handler of the menu item.

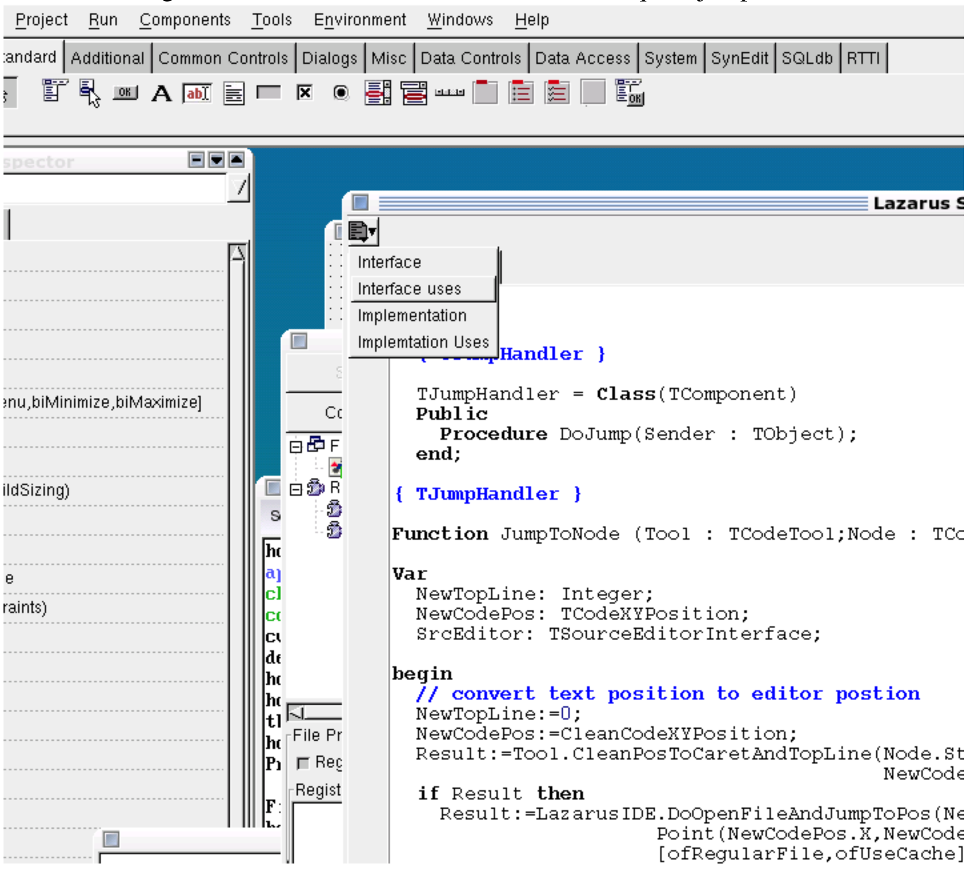
And with that, everything is set. Compiling and installing the package should result in a modified editor window, which looks like figure figure 2 on page 9.

5 Conclusion

In this article, it was shown how to use the code tools. Not many functions were used, but to describe all possible functions of the IDE code tools would probably fill a complete book. Instead, a simple example was created, and the locations of the various functions were shown. The reader wishing to extend the IDE should be able to find the functions he needs on the basis of the indications given here. At the same time, it was shown that the IDE source editor window can be easily extended with custom functionality.

The whole unit to do this contains less than 200 lines, and is understandable for everybody, no detailed knowledge of the IDE is needed. All that is required is ome curiosity to browse the CodeTools package.

Figure 2: The IDE source editor window with quick jump button



```
     pHandler }

  TJumpHandler = Class(TComponent)
  Public
    Procedure DoJump(Sender : TObject);
  end;

{ TJumpHandler }

Function JumpToNode (Tool : TCodeTool;Node : TCc

Var
  NewTopLine: Integer;
  NewCodePos: TCodeXYPosition;
  SrcEditor: TSourceEditorInterface;

begin
  // convert text position to editor postion
  NewTopLine:=0;
  NewCodePos:=CleanCodeXYPosition;
  Result:=Tool.CleanPosToCaretAndTopLine(Node.St
                                        NewCode
  if Result then
    Result:=LazarusIDE.DoOpenFileAndJumpToPos(Ne
                              Point(NewCodePos.X,NewCode
                              [ofRegularFile,ofUseCache]
```