Extending the Lazarus IDE: Menus and the Source editor.

Michaël Van Canneyt

October 1, 2005

Abstract

In previous articles about the lazarus IDE, it was shown how to implement custom projects and files, using the implementation of project templates. In this article, this functionality is extended by adding the templates directly in the menu, and by moving the configuration dialog also to the menu. Furthermore, the workings of the editor and shortcut-key mechanisms are demonstrated by plugging a code formatter in the IDE source editor.

1 Introduction

In previous articles, it was shown how to extend the **'File|New'** menu item dialog. In this article, it is shown how to extend the menu of lazarus itself using the menu interface: Lazarus allows access to all the menus in lazarus, indeed all menus are built using the interface.

The second part of this article shows how to manipulate the text in the IDE source editor. The FCL contains a configurable pascal source formatter. This formatter is used to implement a IDE command which formats a source file or just the selection in the IDE, plus a configuration dialog for the formatter is inserted in the menu.

2 The lazarus IDE Menu Interface

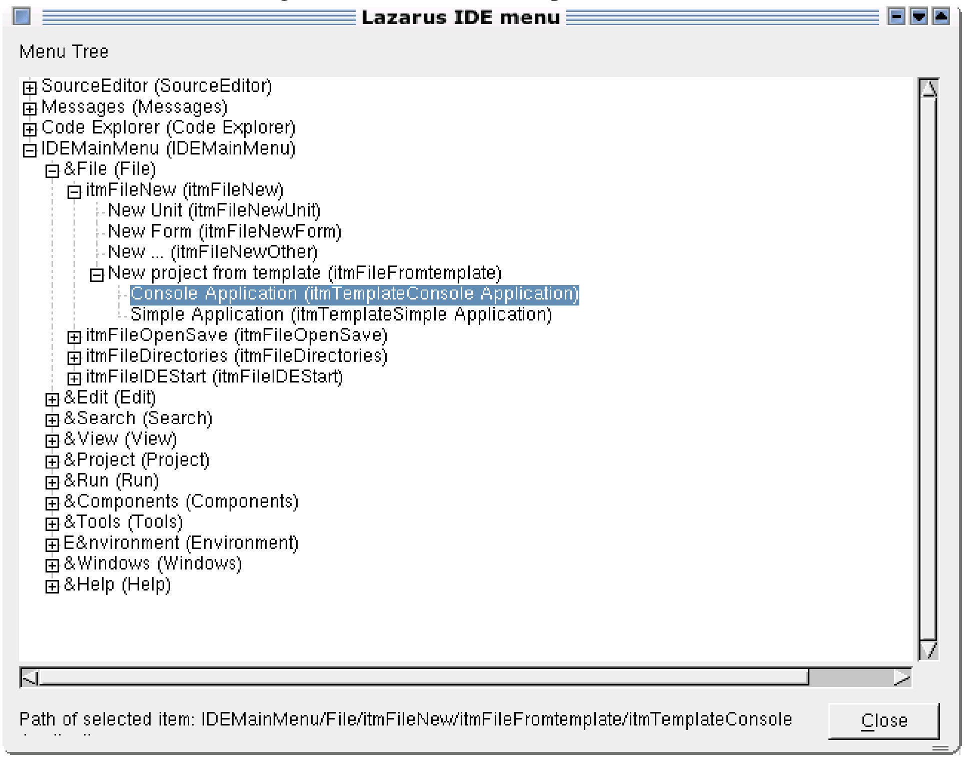
The lazarus IDE menu interface is built around 3 concepts:

Menu Roots The menu roots are the various places in the IDE where a menu occurs. This can be the main menu, but also the various popup menus used in e.g. the source editor or the messages window. The menu roots are managed by the `TIDEMenuRoots` class. It simply keeps a list of menu root locations.

Menu Sections A menu section is a group of menu items. This can be a complete menu (the main menu), or a submenu, e.g. the **'File'** menu item, but it can also be a number of items separated by a separator menu, e.g. the various **'New'** items in the **'File'** submenu. The class `TIDEMenuSection` represents a menu section.

Menu Commands The menu commands are the actual menu items in a menu section. They have an 'OnClick' event, which allows to do whatever is needed when the menu item is selected. It's also possible to assign shortcuts to the menu item through the IDE commands interface. The class `TIDEMenuCommand` represents a menu item.

Figure 1: The IDE menu explorer window



```
⊞ SourceEditor (SourceEditor)
⊞ Messages (Messages)
⊞ Code Explorer (Code Explorer)
⊟ IDEMainMenu (IDEMainMenu)
  ⊟ &File (File)
    ⊟ itmFileNew (itmFileNew)
        New Unit (itmFileNewUnit)
        New Form (itmFileNewForm)
        New ... (itmFileNewOther)
      ⊟ New project from template (itmFileFromtemplate)
          Console Application (itmTemplateConsole Application)
          Simple Application (itmTemplateSimple Application)
    ⊞ itmFileOpenSave (itmFileOpenSave)
    ⊞ itmFileDirectories (itmFileDirectories)
    ⊞ itmFileIDEStart (itmFileIDEStart)
  ⊞ &Edit (Edit)
  ⊞ &Search (Search)
  ⊞ &View (View)
  ⊞ &Project (Project)
  ⊞ &Run (Run)
  ⊞ &Components (Components)
  ⊞ &Tools (Tools)
  ⊞ E&nvironment (Environment)
  ⊞ &Windows (Windows)
  ⊞ &Help (Help)

Path of selected item: IDEMainMenu/File/itmFileNew/itmFileFromtemplate/itmTemplateConsole
```

In general, registering a new menu item is a matter of 2 steps:

1. Implement a method that can be called when the menu item is chosen. This can be a method when the code to be executed is part of a class, or a simple procedure when there is no class.
2. The menu item is registered in an existing menu section with the above procedure/method as a callback.

There are 2 optional steps which can be necessery before the above steps can be performed:

- In case a new menu section is desirable, there is an extra step which must be done before these 2 steps: a menu section must be registered.
- In case there is a complete new menu (usually a popup menu), a menu root must be registered first, before registering a section.

The menu sections are organized hierarchically. It's possibly to find a menu section or menu item by specifying a path. The path is simply the names of all menu sections that must be traversed, separated by a slash (/) character. The epxloreidemenu package, which can be installed in the IDE, shows the complete menu structure as it is known by the IDE. It registers a new menu item under the **'Tools'** menu: **'Explore IDE menu'**. Clicking this item will show the dialog as in figure 1 on page 2. Selecting an item shows the path of the selected item: in the dialog, the menu roots are clearly visible, and the sections are clearly marked.

The code for the IDE menu explorer window is extremely simple. In the `OnCreate` method of the form, the tree-view is filled with the menu structure:

```
procedure TExploreIDEMenuForm.FormCreate(Sender: TObject);

Var
  I : Integer;

begin
  With TVIDEMenu.Items do
    begin
    BeginUpdate;
    Try
      Clear;
      For I:=0 to IDEMenuRoots.Count-1 do
        AddMenuItem(Nil,IDEMenuRoots[I]);
    Finally
      EndUpdate;
    end;
    end;
end;
```

The `TVIDEMenu` is the `TTreeView` component. The `IDEMenuRoots` component is defined in menuintf, and looks as follows:

```
TIDEMenuRoots = class(TPersistent)
  procedure RegisterMenuRoot(Section: TIDEMenuSection);
  procedure UnregisterMenuRoot(Section: TIDEMenuSection);
  function Count: Integer;
  procedure Clear;
  procedure Delete(Index: Integer);
  function IndexByName(const Name: string): Integer;
  function FindByName(const Name: string): TIDEMenuSection;
  function CreateUniqueName(const Name: string): string;
  function FindByPath(const Path: string;
                      ErrorOnNotFound: boolean): TIDEMenuItem;
public
  property Items[Index: integer]: TIDEMenuSection;
end;
```

This is a simple component which keeps a list of `TIDEMenuSection` instances: The various menu's in the IDE. The component has some methods for registering and unregistering menu sections, as well as some methods for searching.

The code to fill the treeview simply calls `AddMenuItem` with all the items in the `IDEMenuRoots` instance. The `AddMenuItem` method is also a quite simple recursive procedure:

```
procedure AddMenuItem(ParentNode : TTreeNode;
                      Item : TIDEMenuItem);

Var
  N : TTreeNode;
  I : Integer;
  Sec : TIDEMenuSection;
  S : String;

begin
  With Item do
```

```
    begin
    S:=Format('%s (%s)',[Caption,Name]);
    N:=TVIDEMenu.Items.AddChild(ParentNode,S);
    N.Data:=Item;
    end;
  if Item is TIDEMenuSection then
    begin
    Sec:=(Item as TIDEMenuSection);
    For I:=0 to Sec.Count-1 do
      AddMenuItem(N,Sec.Items[I]);
    end;
end;
```

It constructs a text for a node in the tree, and adds the node to the tree. It stores a reference to `Item` in the associated data for the Node.

After that, it checks whether the `Item` is of class `TIDEMenuSection`. If this is the case, it calls itself recursively for each subitem in the section, passing the newly created node as the parent for the new nodes.

This makes clear that `TIDEMenuSection` and `TIDEMenuCommand` are both descendents of `TIDEMenuItem`. `TIDEMenuItem` is the equivalent of `TMenuItem`, forming a tree which is subclassed in branch nodes (the sections) and leaf nodes (the commands).

The `TIDEMenuItem` class closely resembles the well-known `TMenuItem` class, as can be seen in the following (partial) declaration:

```
TIDEMenuItem = class(TPersistent)
  property Name: string;
  property Bitmap: TBitmap;
  property Hint: String;
  property ImageIndex: Integer;
  property Visible: Boolean;
  property OnClick: TNotifyEvent;
  property OnClickProc: TNotifyProcedure;
  property Caption: string;
  property Section: TIDEMenuSection;
  property Enabled: Boolean;
  property MenuItem: TMenuItem;
  property MenuItemClass: TMenuItemClass;
  property SectionIndex: Integer;
  property AutoFreeMenuItem: boolean;
  property Tag: Integer;
end;
```

The properties should speak for themselves. Note the reference to `MenuItem`, which refers to the actual menu item in the Lazarus menu.

The `TIDEMenuSection` class is a simple descendent of this class:

```
TIDEMenuSection = class(TIDEMenuItem)
  property ChildsAsSubMenu: boolean;
  property SubMenuImages: TCustomImageList;
  property Items[Index: Integer]: TIDEMenuItem;
  property TopSeparator: TMenuItem;
  property BottomSeparator: TMenuItem;
  property NeedTopSeparator: boolean;
  property NeedBottomSeparator: boolean;
  property VisibleCount: integer;
  property States: TIDEMenuSectionStates;
```

The `NeedTopSeparator` and `NeedBottomSeparator` properties determine the presence of a separator menu item at the beginning and end of the section. The `TopSeparator` and `BottomSeparator` properties contain the actual items created by the IDE. The `ChildsAsSubmenu` property determines whether the children of this section are shown in a submenu or not.

To create the menu item for the menu explorer, the following code is inserted in the `Register` procedure of the unit:

```
Procedure Register;

begin
  RegisterIDEMenuCommand(itmSecondaryTools,
                         SExploreIDEMenu,
                         SExploreIDEMenuCaption,
                         Nil,
                         @ShowMenu,
                         Nil);
end;
```

The `SExploreIDEMenu` and `SExploreIDEMenuCaption` are two string constants. The `ShowMenu` method is explained below.

The `RegisterIDEMenuCommand` command is defined as follows:

```
function RegisterIDEMenuCommand(Parent: TIDEMenuSection;
                                Name, Caption: string;
                                OnClickMethod: TNotifyEvent;
                                OnClickProc: TNotifyProcedure;
                                Command: TIDECommand
                                ): TIDEMenuCommand;
function RegisterIDEMenuCommand(Path, Name, Caption: string;
                                OnClickMethod: TNotifyEvent;
                                OnClickProc: TNotifyProcedure;
                                Command: TIDECommand
                                ): TIDEMenuCommand;
```

The meaning of the arguments in this call should be obvious:

Parent The menu section to which the menu item should be appended.

Path As an alternative to `Parent`, the path to the section can be specified.

Name a name for the menu item.

Caption a caption for the menu item.

OnClickMethod Method called when the menu item is clicked. (optional)

OnClickProc Procedure called when the menu item is clicked. (optional)

Command A `TIDECommand` class, for a shortcut key (optional)

The `itmSecondaryTools` is one of the many standard IDE menu sections. All standard menu sections are defined as variables starting with `itm` in the menuintf unit, and point to instances created by the IDE. They can be used to insert menu items in the various parts of the IDE menu. The IDE menu explorer is registered in a section of the **'Tools'** menu.

The code to show the actual epxlorer form is quite simple and is given only for completeness:

```
Procedure ShowMenu(Sender : TObject);

begin
  With TExploreIDEMenuForm.Create(Application) do
    try
      ShowModal;
    Finally
      Free;
    end;
end;
```

3 Extending the Project Templates package

The previous section showed how to extend the IDE menu by adding a simple command to the IDE menu. This section will show how to make use of a section. To this end, the 'Project Templates' package presented in previous articles, will be extended in 2 ways:

1. The 'Configuration' dialog will be inserted under the **'Tools'** menu.
2. A new submenu will be created under the **'File'** menu: **'New from template'**. In this submenu, an item will be created for each template in the template directory.

For more information about this package, please refer to the 2 previous issues of Toolbox.

The unit that needs to be changed in the project templates package is idetemplateproject. The first of the two changes is not so difficult, it resembles the code presented in the previous section. The following code is added to the `Register` routine:

```
RegisterIdeMenuCommand(itmCustomTools,
                       STemplateSettings,
                       SProjectTemplateSettings,
                       nil,@ChangeSettings);
```

The `ChangeSettings` method is quite simple:

```
procedure ChangeSettings(Sender : TObject);

begin
  With TTemplateSettingsForm.Create(Application) do
    Try
      Templates:=IDETemplates;
      if ShowModal=mrOK then
        begin
        SaveTemplateSettings;
        UnRegisterKnownTemplates;
        RegisterKnownTemplates;
        end;
    Finally
      Free;
    end;
end;
```

After passing the templates collection to the settings dialog, and showing the form, the template settings are saved. The current templates are unregistered and the new templates are registered. The two methods for that are discussed below, as they will be changed for the second of the proposed changes.

To be able to add a submenu to the **'File'**, a `TIDEMenuSection` must be registered. This is also done in the `Register` routine:

```
itmFileNewFromTemplate:=
   RegisterIDESubMenu(itmFileNew,
                      'itmFileFromtemplate',
                      SNewFromTemplate);
```

The `RegisterIDESubMenu` call is defined as follows:

```
function RegisterIDESubMenu(Parent: TIDEMenuSection;
                            Name, Caption: string;
                            OnClickMethod: TNotifyEvent;
                            OnClickProc: TNotifyProcedure
                            ): TIDEMenuSection;
function RegisterIDESubMenu(Path, Name, Caption: string;
                            OnClickMethod: TNotifyEvent;
                            OnClickProc: TNotifyProcedure
                            ): TIDEMenuSection;
```

The arguments are similar to the registration of a command, and again the location can be specified by specifying the parent menu section, or by specifying the path. The only difference is that no `IDECommand` can be specified: it's not useful to attach a shortcut key to a submenu.

Note that in the above code, the submenu is appended to the `itmFileNew` section, and that the created `TIDEMenuSection` is saved in a variable called `itmFileNewFromTemplate`. The reason for this is that a reference to the section is needed when registering the templates as menu items in the sub menu. This happens in the `RegisterKnownTemplates` call:

```
procedure RegisterKnowntemplates;

Var
  I : Integer;
  ATemplate : TProjectTemplate;
  ProjDesc : TTemplateProjectDescriptor;
  ProjMenu : TIDEMenuCommand;

begin
  For I:=0 to IDETemplates.Count-1 do
    begin
    Atemplate:=IDETemplates[i];
    ProjDesc:=TTemplateProjectDescriptor.Create(Atemplate);
    RegisterProjectDescriptor(ProjDesc,STemplateCategory);
    ProjMenu:=RegisterIDEMenuCommand(itmFileNewFromTemplate,
                                     SItmtemplate+Atemplate.Name,
                                     ATemplate.Name,
                                     Nil,@DoProject,Nil);
    MenuList.Add(TIDEObject.Create(ProjDesc,ProjMenu));
    end;
end;
```

There are 2 changes in this call, when compared to the previous version:

- In addition to a project descriptor, a menu item is registered as well: this happens with a `RegisterIDEMenuCommand`, passing it the `itmFileNewFromTemplate` as the parent menu section. The template name is used as the caption.
- All registered project descriptors and `TIDEMenuCommand` instances are kept in a list (`MenuList`). This is needed to be able to unregister the items again, and for the `DoProject` callback. The `TIDEObject` is simply an object that has 2 fields to keep the instances. The list is of type `TObjectList`, and will free it's contents when it is cleared.

The `DoProject` click handler is called for each item in the menu. It searches the command in the list of registered commands, and saves the corresponding `TProjectDescriptor`:

```
Procedure DoProject(Sender : TObject);

Var
  I : Integer;
  Desc : TTemplateProjectDescriptor;

begin
  I:=MenuList.count-1;
  Desc:=Nil;
  While (Desc=Nil) and (I>=0) do
    begin
    With TIDEObject(MenuList[i]) do
      if FProjMenu=Sender then
        Desc:=FProjDesc;
    Dec(i);
    end;
  If Desc<>Nil then
    LazarusIDE.DoNewProject(Desc);
end;
```

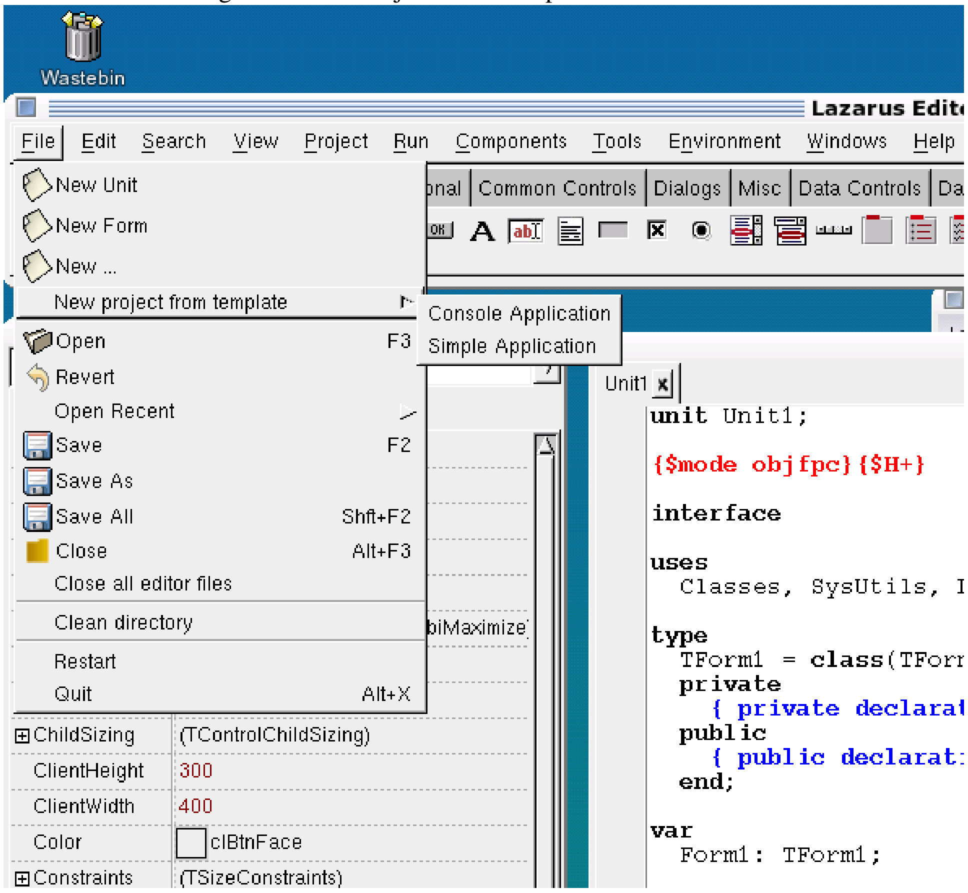
If a project descriptor was found, the `DoNewProject` method of the IDE is called, passing it the descriptor. This will then create a new project based on the chosen template, as if it had been chosen from the **'File|new'** dialog.

When the template directory is changed, the existing templates must be unregistered, and the IDE menu items must be cleaned up. This happens in the `UnRegisterKnownTemplates` procedure, which is actually a simple loop:

```
procedure UnRegisterKnowntemplates;

Var
  I : Integer;
```

Figure 2: The 'Project from Template' menu in action



⊞ChildSizing	(TControlChildSizing)
ClientHeight	300
ClientWidth	400
Color	☐ clBtnFace
⊞Constraints	(TSizeConstraints)

```
unit Unit1;

{$mode objfpc}{$H+}

interface

uses
  Classes, SysUtils, I

type
  TForm1 = class(TFor
  private
    { private declarat
  public
    { public declarat
  end;

var
  Form1: TForm1;
```

```
begin
  For I:=MenuList.Count-1 downto 0 do
    begin
    With TIDEObject(MenuList[i]) do
      begin
      ProjectDescriptors.UnregisterDescriptor(FProjDesc);
      FreeAndNil(FProjMenu);
      end;
    MenuList.Delete(I);
    end;
end;
```

The loop just unregisters all project descriptors in the list, and frees the associated menu items. Freeing the menu item will simply remove it from the menu, there is no need to 'unregister' the menu item.

The result of the whole operation can be seen in figure 2 on page 9

4 The Lazarus Commands interface

In the previous section, a short reference was made to a `TIDECommand` class. This class is part of the Lazarus IDE commands interface, in the unit idecommands.

The lazarus commands interface exists solely to register shortcuts in the IDE. The shortcuts are special key combinations that exist in the IDE. They are the key combinations shown in the **'Environment|Editor options'** dialog, on the tab 'Key mappings'.

For the source formatter demonstration of the editor interface, two shortcut-keys will be created: one to pretty-print the entire file, one to pretty-print just the selection. This registration is again done in the `Register` procedure of the source-formatter package. (prettyformat).

To register a key, it's necessary to understand how the keys are organized. The keys are organized in 2 ways:

Scopes The scope defines where the key can be used. The IDE defines 3 scopes: The editor as a whole, the designer and the source editor only.

Categories The categories are just a visual aid for the 'Key Mappings' dialog, they have no meaning or function in the IDE itself.

A scope is represented by a `TIDECommandScope` class, which has the following definition:

```
TIDECommandScope = class
public
  function IDEWindowClassCount: integer;
  function CategoryCount: integer;
Public
  property Name: string read FName;
  property IDEWindowClasses[Index: integer]: TCustomFormClass;
  property Categories[Index: integer]: TIDECommandCategory;
end;
```

The `IDEWindowClasses` is a list of `TCustomForm` class references: this is used to determine which windows should react to a short cut key. The `Categories` property are the categories of keys that are defined for this scope.

A reference to the 3 standard scopes of the Lazarus IDE can be found in the following variables:

```
IDECmdScopeSrcEdit
IDECmdScopeSrcEditOnly
IDECmdScopeDesignerOnly
```

They are created and filled by the IDE.

A command category is represented by the `TIDECommandCategory` class, defined as:

```
TIDECommandCategory = class
  property Name: string;
  property Description: string;
  property Parent: TIDECommandCategory;
end;
```

From the definition above, it can be seen that categories are no more than an organizational class: they can be organized in a tree.

All categories and commands are listed in the `TIDECommands` class:

```
TIDECommands = class
  function FindIDECommand(ACommand: word): TIDECommand;
  function CreateCategory(Parent: TIDECommandCategory;
                          const Name, Description: string;
                          Scope: TIDECommandScope = nil):
                          TIDECommandCategory;
  function CreateCommand(Category: TIDECommandCategory;
                         const Name, Description: string;
                         const TheShortcutA,
                         TheShortcutB: TIDEShortCut
                         ): TIDECommand;
  function CategoryCount: integer;
  property Categories[Index: integer]: TIDECommandCategory;
end;
```

The meaning of the various methods and properties should be obvious. The idecommands unit contains an instance `IDECommandList` which allows to retrieve and create command categories and commands.

Finally, a command itself is represented by the `TIDECommand` class:

```
TIDECommand = class
  property Name: String;
  property LocalizedName: string;
  property ShortcutA: TIDEShortCut;
  property ShortcutB: TIDEShortCut;
end;
```

The meanings of these properties should be clear. From the above definition, it should be clear that each command can have up to 2 shortcut sequences. The `TIDEShortcut` is a shortcut definition, it is a record which contains the definition of up to 2 keystrokes:

```
TIDEShortCut = record
  Key1: word;
  Shift1: TShiftState;
  Key2: word;
  Shift2: TShiftState;
end;
```

The `Key` and `Shift` fields are of the same type as the parameters of an `OnKeyDown` or `OnKeyUp` event. If only one keystroke is required (such as `CTRL-C`), then the second key can be set to `VK_UNKNOWN`. For a 2-stroke command such as `CTRL-Q F` (find) the first key contains the `CTRL-Q`, the second `F`.

The function `IDEShortCut` creates a shortcut key definition:

```
function IDEShortCut(Key1: word;
                     Shift1: TShiftState;
                     Key2: word  = VK_UNKNOWN;
                     Shift2: TShiftState = []): TIDEShortCut;
```

This can be used to quickly fill a record.

Putting all this together, the required steps to register a key command are:

1. Register a scope. This step is optional, and should rarely be used.
2. Register a category within a scope. This step is also optional, an existing category can be used.
3. Register a IDEShortcut within a category.

For the code formatter, 2 key sequences will be defined, one to format the selection, one to format the whole file. This is done in the `Register` procedure of the IDE:

```
Var
  CmdFormatSelection : TIDECommand;
  CmdFormatFile      : TIDECommand;

Procedure Register;

Var
  Key : TIDEShortCut;
  Cat : TIDECommandCategory;

begin
{$ifndef USECustomCategory}
  Cat:=IDECommandList.CreateCategory(Nil,
                                     SCatFormatting,
                                     SDescrFormatting,
                                     IDECmdScopeSrcEditOnly);
{$else}
  Cat:=Nil;
  With IDECommandList do
    begin
    I:=CategoryCount-1;
    While (Cat=Nil) and (I>=0) do
      if CompareText(Categories[i].Name,'Custom')=0 then
        Cat:=Categories[i]
      else
        Dec(I);
    end;
{$endif}
  Key:=IDEShortCut(VK_F,[SSctrl,ssShift],VK_UNKNOWN,[]);
  CmdFormatSelection:=RegisterIDECommand(Cat,
                                         SCmdPFSelection,
                                         SDescrPFSelection,
                                         Key);
  Key:=IDEShortCut(VK_F,[SSctrl,ssAlt],VK_UNKNOWN,[]);
  CmdFormatFile:=RegisterIDECommand(Cat,
                                    SCmdPFFile,
                                    SDescrPFFile,
                                    Key);
```

The code works with a conditional define `USECustomCategory`. If it is defined, then the 'Custom' command category is searched, and they keys are defined in this category. If it is not defined, a new category is created in the `IDECmdScopeSrcEditOnly` scope, which means it can only be used in the editor window. Then a `CTRL-SHIFT-F` key seqence is defined to format the selection. `CTRL-ALT-F` is used to format the whole file.

These commands are then used to make entries in the global **’Edit’** menu, and the source editor popup menu, under the **’Refactoring’**. section. This code is similar to the code presented above, with the difference that now a IDECommand is used as well:

```
  RegisterIDEMenuCommand(SrcEditSubMenuRefactor,
                         SCmdPFSelection,
                         SDescrPFSelection,
                         Nil,@PrettyPrintSelection,CmdFormatSelection);
  RegisterIDEMenuCommand(SrcEditSubMenuRefactor,
                         SCmdPFFile,
                         SDescrPFFile,
                         Nil,@PrettyPrintFile,CmdFormatFile);
  RegisterIDEMenuCommand(itmEditBlockIndentation,
                         SCmdPFSelection,
                         SDescrPFSelection,
                         Nil,@PrettyPrintSelection,CmdFormatSelection);
  RegisterIDEMenuCommand(itmEditBlockIndentation,
                         SCmdPFFile,
                         SDescrPFFile,
                         Nil,@PrettyPrintFile,CmdFormatFile);
```

The `SrcEditSubMenuRefactor` variable contains the menu section in the source editor popup menu with the refactoring commands. The `itmEditBlockIndentation` stands for the part of the **’Edit’** menu that contains the (de)indent commands. Other sections could have been used.

The `PrettyPrintFile` and `PrettyPrintSelection` routines are discussed in the next section.

5 The Lazarus Source Editor interface

An interface to the lazarus source editor is exposed in the srceditorintf unit. It contains 2 classes:

TSourceEditorInterface This represents a source editor: one tab in the editor window. It presents commands and properties to view and manipulate the source edited there.

TSourceEditorWindowInterface This represents the editor window of the Lazarus IDE. It exposes the currently active editor.

The variable `SourceEditorWindow` is set by the IDE and contains a reference to an instance of `TSourceEditorWindowInterface`, which is defined as follows:

```
TSourceEditorWindowInterface=class
public
  property ActiveEditor: TSourceEditorInterface;
end;
```

The `TSourceEditorInterface` class is the actual editor. It has the following properties:

```
TSourceEditorInterface = class
public
  property BlockBegin: TPoint;
  property BlockEnd: TPoint;
  property CursorScreenXY : TPoint;
  property CursorTextXY: TPoint;
  property EditorControl: TWinControl;
  property FileName: string;
  property Lines: TStrings;
  property CurrentLineText: string;
  property ReadOnly: Boolean;
  property Selection: string;
  property SelEnd: Integer;
  property SelStart: Integer;
  property SourceText: string;
  property TopLine: Integer;
end;
```

Note the use of `TPoint`: this is a coordinate (col,row) in the source or in the window. The meaning of these properties should be more or less clear:

BlockBegin The start of the selection block.

BlockEnd The end of the selection block.

CursorScreenXY The position of the cursor on the screen.

CursorTextXY The position of the cursor in the source.

EditorControl The actual widget used in the editor. This should normally be a `TSynEdit` control.

FileName The filename of the file being edited.

Lines The content of the source file as `TStrings`.

CurrentLineText The content of the current line.

ReadOnly Set to `True` if the contents of the file are read-only.

Selection The text of the current selection.

SelEnd The start position of the selection (in characters).

SelStart The end position of the selection (in characters).

SourceText The complete text of the source as one string.

TopLine The number of the first visible line on the screen.

The editor contains a lot of methods to manipulate the text:

```
// Text
function SelectionAvailable: boolean;
function GetText(OnlySelection: boolean): string;
procedure SelectText(const StartPos, EndPos: TPoint);
procedure ReplaceLines(StartLine, EndLine: integer;
                       const NewText: string);
```

```
procedure ReplaceText(const StartPos, EndPos: TPoint;
                      const NewText: string);
procedure CopyToClipboard;
procedure CutToClipboard;

// screen and text position mapping
function LineCount: Integer;
function TextToScreenPosition(const Position: TPoint): TPoint;
function ScreenToTextPosition(const Position: TPoint): TPoint;

// characters and pixels
function WidthInChars: Integer;
function HeightInLines: Integer;
function CharWidth: integer;
function CursorInPixel: TPoint;

// update
procedure BeginUndoBlock;
procedure EndUndoBlock;
procedure BeginUpdate;  // block painting
procedure EndUpdate;

// search and replace
function SearchReplace(const ASearch, AReplace: string;
                 SearchOptions: TSrcEditSearchOptions): integer;
```

The meaning of these functions should be obvious. Note that only raw text functions are available: the code tools, which analyze the text in the source editor are not part of the source editor interface. Currently, the code tools are not exposed in an IDE interface.

For our code formatter, we need only 2 properties and a function from this whole list of possibilities. The `PPrettyPrintSelection` callback for our menu command is implemented using 2 of those:

```
Procedure PrettyPrintSelection(Sender: TObject);

Var
  S1,S2 : TSTringStream;
  E : TSourceEditorInterface;

begin
  E:=SourceEditorWindow.ActiveEditor;
  If (E=Nil) or (Not E.SelectionAvailable) then
    Exit;
  S1:=TStringStream.Create(E.Selection);
  Try
    S2:=TStringStream.Create('');
    Try
      S1.Position:=0;
      PrettyPrintStream(S1,S2);
      E.Selection:=S2.DataString;
    Finally
      S2.Free;
    end;
  Finally
```

```
    S1.Free;
  end;
end;
```

The routine is pretty basic: first, a reference to the current editor is obtained, and checked for availability of a selection. The current selection is written to a stream, and then passed to the code formatting routine. After this is done, a second stream contains the formatted code. The selection is then replaced with the formatted code.

Similarly, the whole file can be formatted:

```
Procedure PrettyPrintFile(Sender : TObject);

Var
  S1,S2 : TMemoryStream;
  E   : TSourceEditorInterface;

begin
  E:=SourceEditorWindow.ActiveEditor;
  If (E=Nil) then
    Exit;
  S1:=TMemoryStream.Create;
  Try
    E.Lines.SaveToStream(S1);
    S1.Position:=0;
    S2:=TMemoryStream.Create;
    Try
      PrettyPrintStream(S1,S2);
      S2.Position:=0;
      E.Lines.LoadFromStream(S2);
    Finally
      S2.Free;
    end;
  Finally
    S1.Free;
  end;
end;
```

Instead of the selection, the `Lines` property is used to wrote the whole source file to a stream, and to load it from the result stream.

The `PrettyPrintStream` function uses a `TPrettyPrinter` class (delivered standard with FPC) to do the actual formatting:

```
Procedure PrettyPrintStream(SIn,SOut : TStream);

Var
  PP : TPrettyPrinter;

begin
  PP:=TPrettyPrinter.Create;
  Try
    PP.Source:=Sin;
    PP.Dest:=Sout;
    PP.PrettyPrint;
  Finally
    PP.Free;
  end;
end;
```

At this moment, no options are set, the defaults are used. For example, the following code:

```
procedure TForm1.FormCreate(Sender: TObject);

Var I,J : integer;

begin
  For I:=0 to 100 do begin
    Try
      J:=99 mod I;
      except
        // Do something
      end;
  end;
end;
```

Is replaced with:

```
Procedure TForm1.FormCreate(Sender: TObject);

Var I,J : integer;

Begin
  For I:=0 To 100 Do
    Begin
      Try
        J := 99 Mod I;
      Except
        // Do something
      End;
    End;
End;
```

Note the capitalization and indentation.

The behaviour of the `TPrettyPrinter` class is customizable. However, it would lead too far to discuss all options of this class. The interested reader can have a look at the sources in the ptopu unit.

6 Conclusion

In this article, 3 parts of the Lazarus IDE interfaces have been studied: The shortcut-key mechanism (commands), the menu interface, and the source editor interface. An existing Pretty-Printer component was used to do the actuall formatting. It is not yet finished, for instance a dialog to set the (extensive) options of the pretty-printer could be created and added to the menu. The last great part of the IDE interface, the code tools, are not yet integrated in the IDE interfaces. As soon as this is the case, an essay about it will follow.