Getting started with Lazarus: Programming actions

Michaël Van Canneyt

November 1, 2006

Abstract

A responsive GUI interface is part of every user-friendly program. Separating GUI logic from application logic is something which ensures extensibility and good program design. Actions provide the means to achieve both: they make the user-experience better, and help in splitting the user interface from the application logic.

1 Introduction

Any program should help the user by providing feedback about what the user can or cannot do at any moment. Providing this feedback can mean that a button should be enabled or disabled, an image on a button can be changed, or the action of a button can be changed depending on the data entered by the user. For example, the 'Login' button in a login dialog should not be enabled till at least a user name was entered, and - if a password is required - a password was entered. Likewise, the 'Save' button should not be active until there is something to save: the document was modified or some similar condition.

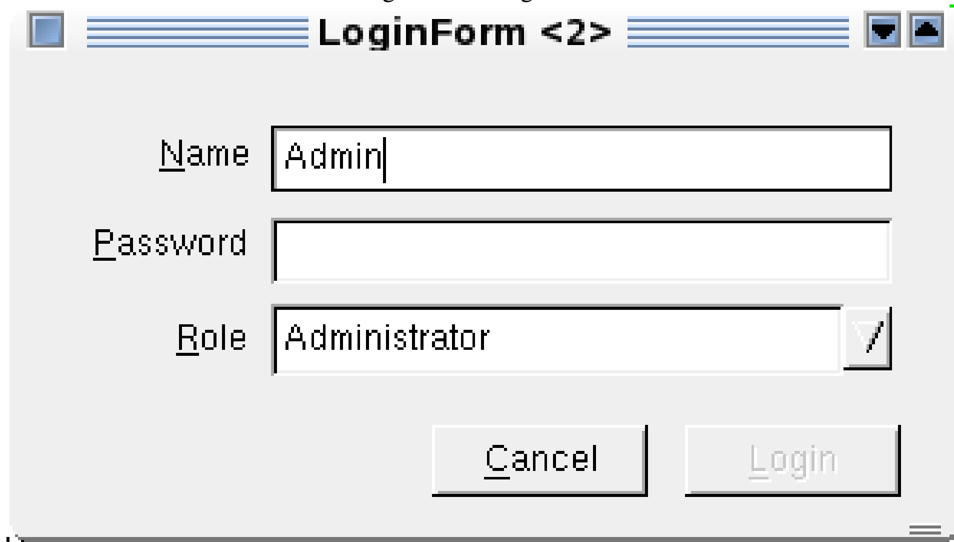
In an event-driven application, this is usually done by updating the status of the various GUI elements when something is changed in a screen. Through the appropriate event handlers, the status of a button can be set. If a lot of data (or controls) needs to be monitored for changes, this can quickly become a whole lot of event handlers. Further more, the event handlers will most likely also be used for other things besides updating the status of other controls.

With Actions, the logic of checking controls and updating the state of other controls can be separated out: The state of an action is checked (or updated) continuously when the application is idle, i.e. the user is not doing anything. This obliviates the need for event handlers in each control that needs to be checked for it's state: one event handler checks all controls at regular intervals.

Most applications provide several means of performing the same thing: Saving a document can be done using a menu entry and a button on a toolbar. Preferably, the menu entry and the toolbar button show the same image, the same hint, and should be enabled or disabled at the same time: quite a lot of properties or event handlers will be duplicated - although the event handlers can be shared.

Here also, actions provide a way to centralize the code: In an action list, all actions that can be performed by the user are gathered and managed. For instance a 'Save' action, an 'Open' action, and so on. The actions can then be associated to several GUI elements: menus, toolbuttons, normal buttons or checkbuttons. As soon as the user activates one of the GUI elements, the action is executed: for some pre-defined actions, the action is performed by the LCL: for example the standard TCutAction will cut the current selection

Figure 1: A Login form



and copy it to the clipboard. For other actions (the standard TAction), a user-defined event is called.

By using actions, the programmer is forced to define clearly what the user actions of a given form are, independent of the GUI elements to execute this action. By grouping them in an action list, the programmer has a clear list of well-defined actions. With some extra coding, the user can be presented with a means of customizing how he wants his actions presented to him: How the menu is composed, and how the toolbars are layed out, and even what shortcuts can be used to execute his actions.

2 The actionless approach

To illustrate how actions make programming easier, it is instructive to compare an action-based form with a form which does not use actions. The example is a simple login form. It has 2 edit boxes: one for a name, one for a password. A dropdown-list with a role is also present, plus 2 buttons: a 'Cancel' and a 'Login' button. It should look like figure 1 on page 2

The login button should be active only when the user has provided a username and password, and has chosen a role. To do this, 4 event handlers must be installed: one for the `OnChange` event of each control, and one in the `OnCreate` event of the form, to initialize the button. If checking the state of the button was the only thing to be done in these events, it would be sufficient to write 1 event handler and assign it to all 4 events, and there would be no gain of using actions. But most event `OnChange` event handlers do also other things: for instance clearing the password if the username is changed. The `OnCreate` handler of the form also checks the command-line, and copies any username/password options found on the command-line to the controls.

All this results in the following code in the program:

```
procedure TLoginForm.ENameChange(Sender: TObject);
begin
  EPassword.Text:='';
  if (EName.Text='Admin') then
```

```
    CBRole.ItemIndex:=2;
  CheckButton;
end;

procedure TLoginForm.CBRoleChange(Sender: TObject);
begin
  CheckButton;
end;

procedure TLoginForm.EPasswordChange(Sender: TObject);
begin
  CheckButton;
end;

procedure TLoginForm.FormCreate(Sender: TObject);

Var
  I : Integer;

begin
  I:=1;
  While I<=ParamCount do
    begin
    If (Paramstr(i)='-u') and (I<ParamCount) then
      begin
      Inc(i);
      EName.Text:=ParamStr(I+1);
      end
    else If (Paramstr(i)='-p') and (I<ParamCount) then
      begin
      Inc(i);
      EPassWord.Text:=ParamStr(I);
      end;
    Inc(I);
    end;
  CheckButton;
end;
```

Note the call to `CheckButton` in each of these event handlers.

The `CheckButton` code checks whether the Login button can be enabled or not:

```
procedure TLoginForm.CheckButton;

begin
  BLogin.Enabled:=(EName.Text<>'') and
                  (EPassword.Text<>'') and
                  (CBRole.ItemIndex<>-1)
end;
```

Obviously, this is a very simple form for demonstration purposes. It could be made even simpler by re-using the event handler `EPasswordChange` for the `CBRole` combobox' `OnChange` event handler. This kind of simplifications is exactly why the use of actions is a good design decision: Supposing that a shared event handler needs to be split up because for one of the controls an additional check needs to be programmed. In that case, the call to

`CheckButton` can be forgotten or even duplicated to places where it is not needed. What is more, the call to `CheckButton` is scattered all over the code. With Actions, this is not so.

Below, the same form will be programmed using Actions.

3 Actions: Architecture

Actions are non-visible components. They are organized in ActionLists, which are just a design-time container. Actions have a series of properties, which correspond to properties often found in visual controls:

Caption The text displayed on the control.

Checked For menus, toolbar- or speedbuttons and checkboxes: whether the item appears checked.

Enabled If the control is enabled or not.

GroupIndex For menu items, checkbuttons or radiobuttons: the group.

HelpContext, HelpKeyword and HelpType The various help possibilities.

Hint The hint displayed in the tooltip.

ImageIndex The ImageIndex in the image list associated with the control (or action list)

ShortCut, SecondaryShortCuts Shortcut keys associated with the action. The action can be executed using one of the shortcuts.

Visible Is the control visible or not ?

OnHint Event handler to retrieve the hint for the control.

OnExecute This handler is executed when the action is activated and needs to execute itself.

OnUpdate This handler is executed in the idle loop of the application: it can be used to update the status of the action depending on the current state of the form.

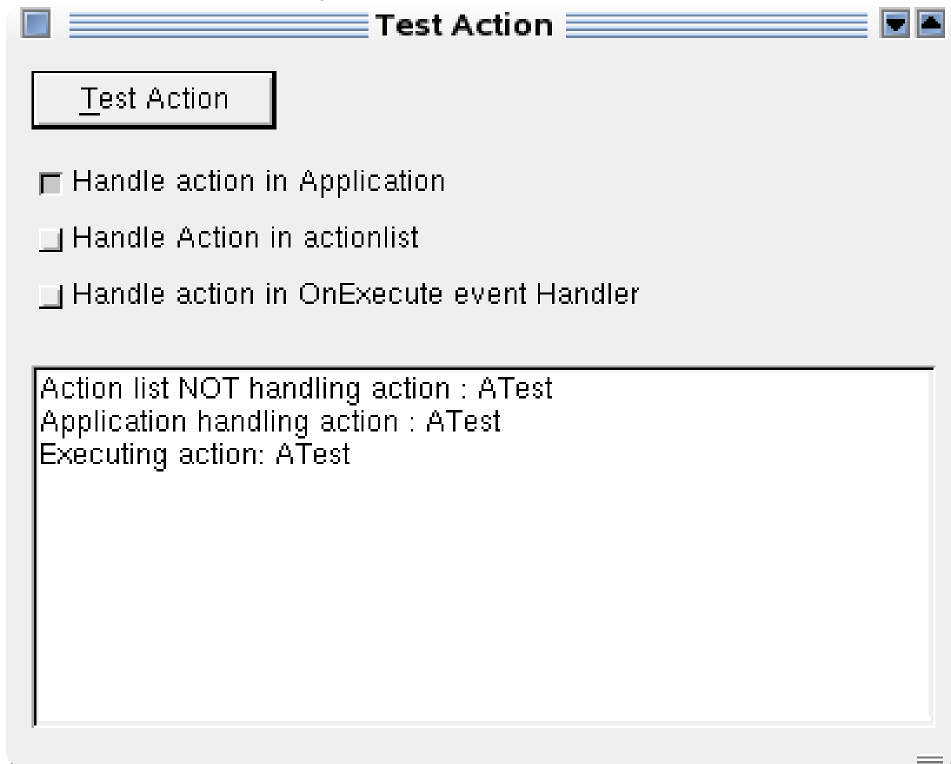
As soon as an action is associated with a control - such a control is called a client of the action - the relevant properties are copied to the control. If any of the properties is changed, the change is instantly propagated to the client controls.

At least the following controls can be action clients: TForm, TButton, TCheckBox, TRadioButton, TToolButton, TSpeedButton, TMenuItem: They all have an 'Action' property. As soon as this property is set, the control is a client of the action. In all cases, as soon as the control is clicked, the corresponding action is executed, which means the `OnExecute` event is executed. Some standard actions perform some well-defined action (copy/cut to clipboard, etc), and do not need an event handler, although one can be specified.

When an action is executed, the following chain of events is executed:

1. The action list of which the action is part is asked to handle the action.
2. If the actionlist didn't handle the action, the application is asked to handle the action.
3. If the application did not handle the action, the action tries to execute itself: typically this means it executes it's `OnExecute` event.

Figure 2: Execution chain test form



By default, an action disables itself if it cannot execute itself: if no `OnExecute` event is set.

To illustrate this, a small test application can be created. It has 3 checkboxes: each of the checkboxes determines at which level the action is handled. A button is associated with an action `ATest`. A log of actions is displayed in a memo. The form is shown in figure 2 on page 5

The action can be handled at the ActionList level by setting the ActionList `OnExecute` handler in the Object Inspector:

```
procedure TMainForm.ActionList1Execute(AnAction: TBasicAction;
                                        var Handled: Boolean);
begin
  Handled:=CBActionList.Checked;
  if Handled then
    begin
    Log('Action list handling action : '+AnAction.Name);
    DoAction(AnAction);
    end
  else
    Log('Action list NOT handling action : '+AnAction.Name);
end;
```

As can be seen from this code, the `Handled` parameter can be used to indicate that the action was handled. If it is not set to `True`, the action is assumed not to be handled, and is handed to the application. A similar action can be set at the application level:

```
procedure TMainForm.ApplicationAction(AnAction: TBasicAction;
                                      var Handled: Boolean);
begin
  Handled:=CBApplication.Checked;
  if Handled then
    begin
    Log('Application handling action : '+AnAction.Name);
    DoAction(AnAction);
    end
  else
    Log('Application NOT handling action : '+AnAction.Name);
end;
```

This handler cannot be set in the Object Inspector, it must be set in code, and this is done in the `OnCreate` handler of the form:

```
procedure TMainForm.FormCreate(Sender: TObject);
begin
  Application.OnActionExecute:=@ApplicationAction;
  ATest.DisableIfNoHandler:=False;
end;
```

The second line sets the `DisableIfNoHandler` property of the Action, because currently it cannot be set in the object inspector. Normally, if a `TAction` does not have a `OnExecute` handler, it disables itself; This is not always desirable, for instance in the example application it has no handler. The handler is only set when the 'CBEvent' checkbox is clicked:

```
procedure TMainForm.CBEventChange(Sender: TObject);
begin
  If CBEvent.Checked then
    ATest.OnExecute:=@DoEventAction
  else
    ATest.OnExecute:=Nil;
end;
```

The other methods of the form (`Log`, `DoEventAction`) are logging methods, and the interested reader can consult the application sources to see them.

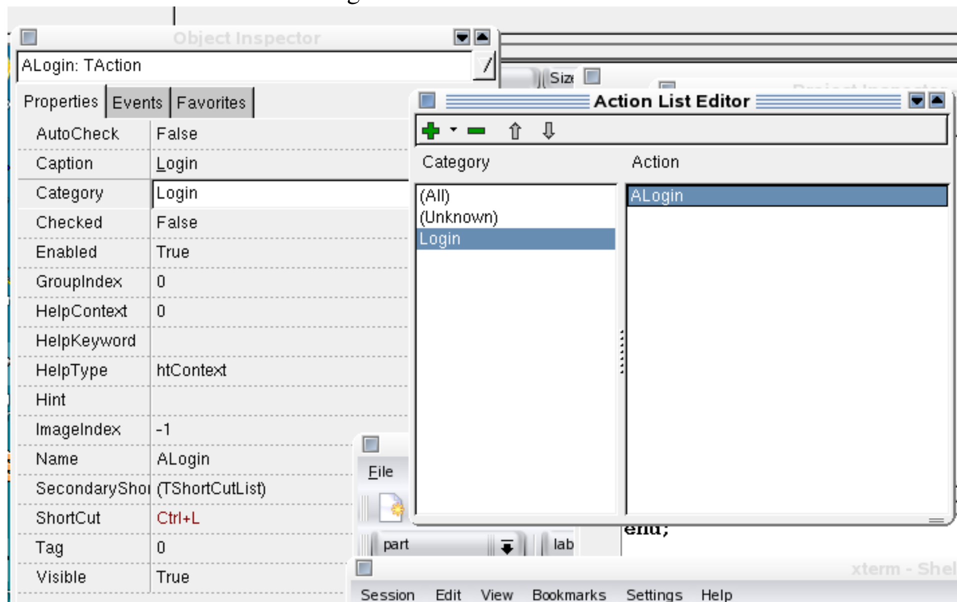
The 3 stage approach is useful in large applications, when it is necessary to provide a consistent interface over the various forms. In such cases it makes sense to handle the actions at the application level, but leave the option to override them at the form level.

4 The login form with actions

Now that the basic structure of actions is clear, the Login form can be re-coded using actions. To do this, an ActionList is dropped on the form. Using a Right-click, the action list editor can be invoked (see figure 3 on page 7). In the editor, a single action can be added, with caption 'Login', and name 'ALogin'. Once this is done, the `Action` property of the 'BLogin' button can be set to the 'ALogin' action in the object inspector. Now the `OnExecute` and `OnUpdate` handlers of the action can be programmed: The `OnExecute` handler is implemented with a dummy login:

```
procedure TLoginForm.ALoginExecute(Sender: TObject);
```

Figure 3: The action list editor




```
begin
  ShowMessage('Logging in');
end;
```

As soon as this is done, one can observe in the Object Inspector that the 'OnClick' handler of the button is set to the `ALoginExecute` method.

The final method to be implemented is the `OnUpdate` handler of the action. This method is called when the application enters the idle state, and should always contain very short or fast code: if too much time is spent updating the action, this will make the application very unresponsive. For the login form, the handler is simple:

```
procedure TLoginForm.ALoginUpdate(Sender: TObject);
begin
  (Sender as TAction).Enabled:=(EName.Text<>'') and
                                (EPassword.Text<>'') and
                                (CBRole.ItemIndex<>-1)
end;
```

Which is almost the `CheckButton` code. The `Enabled` property of the action is propagated to the 'Blogin' button, enabling or disabling it depending on what the user enters.

The `CheckButton` method can be removed, and the calls to it as well. This makes the code of the form simpler and less error prone. Indeed, some of the event handlers can now be removed. For the login form presented here, this is not so much, but for more complex forms, the gain would be more substantial.

5 Creating new actions

By default, a new action is of type `TAction`. This is an action which can be used for all purposes: it's functionality is provided through event handlers such as `OnUpdate` and

`OnExecute`. However, there exist some pre-defined classes which execute well-defined procedures.

There are a lot of pre-defined classes available in Lazarus:

Editing Mostly actions which interact with an edit control: Cut, copy paste the selection, select the full text, Undo and delete actions.

Help Actions to invoke a help system.

Dialog Actions to open font and color dialogs.

File Actions that fit in a file menu, such as open and save.

Database Actions acting on a `TDataset` instance (or `TDatasource`): navigation commands, edit, post, cancel, delete.

For all other actions, either a standard `TAction` can be used, or an own action class can be programmed and registered in the IDE.

Programming a custom action is very easy. Only 3 methods need to be overridden. To demonstrate this, a simple action which clears an edit control will be programmed:

```
TClearAction = Class(TAction)
Public
  function HandlesTarget(Target: TObject): Boolean; override;
  procedure UpdateTarget(Target: TObject); override;
  procedure ExecuteTarget(Target: TObject); override;
end;
```

The first method to be programmed is the `HandlesTarget` method. When updating or executing an action, first the target of the action is determined. This is done by passing a series of controls to the `HandlesTarget` method. The following controls are passed:

1. The currently focused control.
2. The current form.
3. All visible controls on the form.

As soon as an appropriate target is found, the search stops. After a target is found, the `UpdateTarget` or `ExecuteTarget` method is executed. When updating, if no valid target is found, the action is disabled if the `DisableIfNoHandler` property is `True` - which is the default value for `TAction`.

So, for the `TClearAction`, the HandlesTarget should check whether the target is a custom edit, and whether it has focus. This can be done with the following code:

```
function TClearAction.HandlesTarget(Target: TObject): Boolean;
begin
  Result:=(Target is TCustomEdit) and (TCustomEdit(Target).Focused)
end;
```

If no target is found, the action will be disabled. If a target is found, it is still possible to give feedback to the user. In the case of the clear action, the edit control should contain some text. If no text is present, the action should be disabled:

```
procedure TClearAction.UpdateTarget(Target: TObject);
begin
  Enabled:=(TCustomEdit(Target).Text<>'')
end;
```

Lastly, when the action is executed, the `ExecuteTarget` method is called, which should clear the edit control. This can be done as follows:

```
procedure TClearAction.ExecuteTarget(Target: TObject);
begin
  If (Target is TCustomEdit) then
    (Target as TCustomEdit).Clear;
end;
```

The check to see whether `Target` is really a `TCustomEdit` is normally not needed, since `HandlesTarget` should have verified this before the Execute method is called.

To register the action in the IDE, a call to `RegisterActions` can be inserted in the lazarus package. This should look as follows:

```
  RegisterActions('Edit',[TClearAction],Nil);
```

The first parameter is the category in the Action List Editor. The second is a list of action classes one wishes to register (just one in the above example), and the last parameter is a resource component which can contain resources such as images, associated with the classes. This will not be discussed in this article. Instead, the demo program will create the action at runtime.

In the example program, some buttons are dropped on a form, as well as a speedbutton, an edit and memo control. In the `OnCreate` handler of the form, the action is created:

```
procedure TMainForm.FormCreate(Sender: TObject);
begin
  AClear:=TClearAction.Create(Self);
  AClear.Caption:='&Clear';
  AClear.ActionList:=ALTest;
  SBClear.Action:=AClear;
end;
```

The action is assigned to the speedbutton `SBClear`. The reason for making it a speedbutton (and not a regular button) is that a speedbutton does not receive focus when it is clicked. If it were a regular button, the action would never be executed: at the moment of the click, the `HandleTarget` would always return `False`, since the button would have focus, and therefore the action would never be executed.

When the demo application is run, it should look like figure 4 on page 10. The button will be active when the focus is on the edit or memo control, and it has some text in it.

6 Conclusion

In this article, it was shown how to work with actions in Lazarus applications. The advantages of actions were explained, and it was shown how to work with them. It was even shown how to create new custom actions for inclusion in the IDE. One of the benefits of actions - the ability to customize the toolbars and menu's of an application by the user - has not been touched: this may be the subject of an future contribution, as it is a complex topic which would lead too far for an introduction.

Figure 4: The custom action at work