Actions And Action Lists

by Brian Long

Actions and action lists were introduced with Delphi 4 in June 1998 (in all flavours of the product) and were apparently considered so important that they were the first addition to the `Standard` page of the Component Palette since Delphi was released back in February 1995.

However, it would appear that these potentially very useful components have been much under-used by the Delphi community. Maybe this is just because people don't know about them. Maybe the `TActionList` component should have been placed at the beginning of the `Standard` page rather than the end (you may have noticed that in Delphi 5 the `Frames` selector was added at the start of the `Standard` page).

Anyway, for those who have never had the time or inclination to look into what actions do and how they work, this article will explore their purpose, use and internal operation. It closes with a look at how to install new, reusable actions into the IDE.

What Is An Action?

Often in applications, there are several UI mechanisms to trigger the same functionality (or command). For example, a normal button, a menu item and maybe a tool button on a tool bar, or a speedbutton on a speedbar. Normally you set up this type of arrangement by sharing `OnClick` event handlers between the various objects. Of course, you must set up the captions and various other properties of each control individually; this also applies when you need to disable all the controls that can invoke the command.

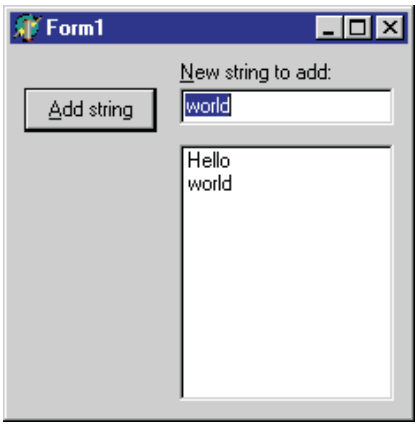
An action is a non-visual component that represents a user-generated command. It allows you to set up all the UI properties related to that command in one place, along with the code needed to execute the command and *also* code that can control whether the command is available to the user or not. Actions are typically managed through action lists, which are also non-visual components.

You connect actions to various trigger controls which can invoke the action. The UI properties and command functionality are both automatically propagated from the action to these controls. If any property of the action is modified at any point, these changes are also propagated. So, for example, any time the action gets disabled, all the related controls are disabled in turn. Code that controls whether the actions are available or not is automatically called during idle time, meaning that UI controls that can invoke the action are automatically enabled and disabled as appropriate.

The trigger controls that are designed to invoke the action's code are called *action clients*. The controls affected by the action are described as *action targets*. Normally, when you create actions in the IDE you do not explicitly specify action targets (there is no place to do so). Instead, the action code implicitly affects various action targets. We will see how action targets gain more significance later.

➤ *Listing 1: Functionality and validation in one place.*

```
procedure TForm1.btnAddStringClick(Sender: TObject);
begin
  if ( Length( Trim( edtEntry.Text ) ) > 0 ) and
    ( lstEntries.Items.IndexOf( Trim( edtEntry.Text ) ) = -1 ) then
    lstEntries.Items.Add( Trim( edtEntry.Text ) );
  //Give focus back to edit
  edtEntry.SetFocus;
  //Highlight edit contents so it can be replaced by overtyping
  edtEntry.SelectAll
end;
```



➤ *Figure 1: The application without actions.*

Why Should We Use Actions?

The answer to this question is simply because they are easier to deal with. They allow application code to be modularised and defined independently of the controls that will invoke the code. Actions are also automatically updated, thereby updating the action clients. The Delphi IDE (from Delphi 4 onwards) is positively chock full of action objects. Actions are responsible for all the tool button and menu controls that become enabled and disabled as circumstances change in the Delphi IDE.

Let's try implementing a simple application a few times: firstly without actions in the normal way, then using actions. Hopefully, you will see that actions simplify the development of application functionality and the management of a smooth UI. The application has an edit control (`edtEntry`), a button (`btnAddString`) and a listbox (`lstEntries`), as shown in Figure 1. The button's job is to add the edit's contents into the listbox. However, it only does this if the edit control does not contain a blank string or a string already contained in the list.

To start with, our attempts will be without actions. There are two approaches we can choose from. The first one is to implement the

```
procedure TForm1.btnAddStringClick(Sender: TObject);
begin
  lstEntries.Items.Add( Trim( edtEntry.Text ) );
  //Give focus back to edit
  edtEntry.SetFocus;
  //Highlight edit contents so it can be replaced by over-typing
  edtEntry.SelectAll;
  //Trigger edit's OnChange to ensure button is enabled/disabled as appropriate
  edtEntry.OnChange( edtEntry )
end;
procedure TForm1.edtEntryChange(Sender: TObject);
begin
  btnAddString.Enabled := ( Length( Trim( edtEntry.Text ) ) > 0 ) and
    ( lstEntries.Items.IndexOf( Trim( edtEntry.Text ) ) = -1 )
end;
```

➤ *Listing 2: The functionality split from the validation.*

button's `OnClick` handler as shown in Listing 1. As you can see, the basic job of adding the text into the listbox is done here, as is the validation of whether to perform the job at all. In this case the button is always enabled, but sometimes pressing it has no effect.

In more complex situations it could prove advantageous to split the implementation of the job from the validation code, as in Listing 2. Here, the validation is performed when the edit content is changed, and also after adding a string into the list. Invalid input in the list is avoided by ensuring the button is disabled when invalid data is in the edit, which perhaps gives a more intuitive user interface (see Figure 2). However, to achieve this, the button must also be disabled in the Object Inspector, since the edit will start its life empty. This code can be found in the ActionLessApp.dpr project on the disk.

As you can see, the validation is *almost* automatic, but not quite. When the text is added to the list, the edit's `OnChange` event must be explicitly invoked to ensure that the button is disabled whilst the edit contains a string contained in the listbox.

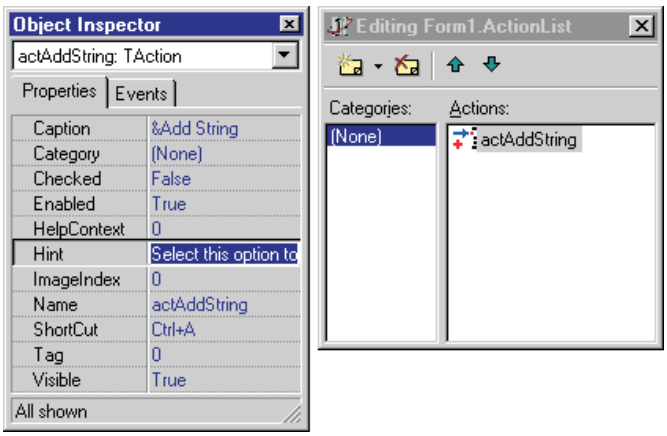
Now think about what would be needed if more controls could invoke the string adding behaviour. You would need to share the button's `OnClick` handler with all the other controls. You would also need to set the `Enabled` property of all the other controls in the edit's `OnChange` event handler. This would quickly get messy to manage and maintain. This is where actions become very useful, although they are just as appropriate when only a single control invokes some behaviour.

How Actions Are Used

You typically set up actions before putting the action client controls on the form. Once the actions are defined, it is then easy to add client controls and associate them with the actions.

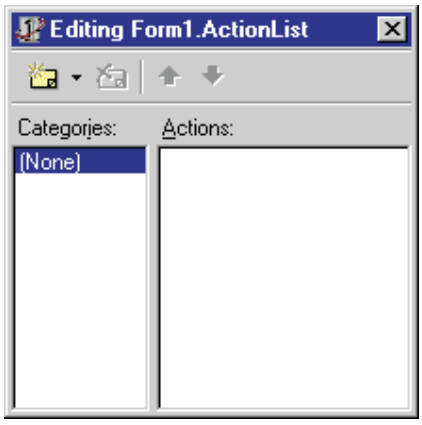
In order to rebuild this application with actions and see how things change, let's get some basic background first. The full details will be covered later.

Actions are managed by action lists (`TActionList` components). You can use as many action lists as you like, perhaps using multiple instances to keep actions in different logical groups. Each action list can be associated with a `TImageList` that contains small images that can optionally be used to represent each action. If you have set up an image list, use the action list's `Images` property to make the connection.



➤ *Figure 4: Editing an action's properties.*

➤ *Figure 2: Prototype number 2.*



➤ *Figure 3: The Action List Editor.*

Creating Actions

Once you place an action list on a form you can add actions using the Action List Editor, available by right clicking or double clicking on it (see Figure 3). This allows you to create new actions and new standard actions (see later for details on standard actions).

Clicking the yellow button (or pressing `Insert`) makes a new action. Whilst selected, the Object Inspector can be used to set up the UI properties that represent the action. Figure 4 shows a new action with a number of properties including a shortcut key (`ShortCut`) and an index into the action list's image list (`ImageIndex`). The action's image is shown in the action list editor.

One property that warrants description is `Category`. All actions in an action list have a category (a

string) but normal actions default to having none. You can choose categories for each of your actions and the action list editor will list all categories in its left listbox. When any given category is selected, only the actions from that category are listed in the right listbox.

The Object Inspector also allows you to set up a number of events for each action, the most important of which are `OnExecute` and `OnUpdate`. `OnExecute` should perform the job represented by the action. `OnUpdate` should verify whether the action is still valid.

Suitable action event handlers for an action that can work in our application are shown in Listing 3 (they are in ActionApp.dpr on the disk). Notice that this time, unlike with Listing 2, the string adding code does not need to trigger the validation code. Instead it relies on the code being automatically called, which it will be (again, the details are coming soon).

At this stage we have not specified any action clients, although the code implicitly identifies action targets of the edit control and listbox. However, these are hard-coded in source and so do not quite fit the normal definition of action targets (more on action targets later).

Invoking Actions

We need a way to invoke the action. Normal procedure would involve adding action clients and connecting the action to them, but this is *not* strictly necessary. The action has already defined a shortcut (`Ctrl+A`). At runtime, pressing `Ctrl+A` will automatically invoke the action, if it is available (in other words is enabled), which frankly is rather clever.

```
procedure TForm1.actAddStringExecute(Sender: TObject);
begin
  lstEntries.Items.Add( Trim( edtEntry.Text ) );
  //Give focus back to edit
  edtEntry.SetFocus;
  //Highlight edit's content so it can be replaced by over-typing
  edtEntry.SelectAll;
end;
procedure TForm1.actAddStringUpdate(Sender: TObject);
begin
  (Sender as TAction).Enabled := ( Length( Trim( edtEntry.Text ) ) > 0 ) and
    ( lstEntries.Items.IndexOf( Trim( edtEntry.Text ) ) = -1 )
end;
```

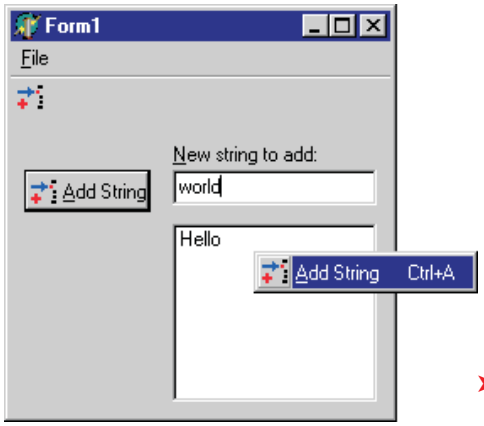
➤ *Listing 3:*
Action event handlers.

In this application though, action clients are required. Drop a button on the form and use the `Action` property to connect it to the only available action, `actAddString`. It will immediately absorb all appropriate properties and events from the action, which are `Caption`, `Enabled`, `HelpContext`, `Hint`, `Visible` and `OnExecute` (assigned to `OnClick`). These make the button look and behave sensibly. When the button is clicked, the action will be invoked.

Notice that the action's properties and events do not necessarily tie up with similarly named properties and events in the client. This allows more flexibility in associating actions with a whole variety of action clients.

If the default action client invocation mechanism is not suitable (say, for example, you want a control double-clicked to invoke the action, or the control has no `Action` property) this is no problem. You can either make the control's favoured event share the action's `OnExecute` event handler, if compatible, or you can make an explicit call to the action's `Execute` method in the control's event handler.

The action's `OnUpdate` event is regularly called during application idle time (again, full details coming later) and so if any of the validation conditions fail, the action's `Enabled` property is set to `False`. This change propagates to the button, causing it to become disabled, meaning that when the action is disabled the action client cannot trigger the action. The program running looks very much like Figure 2.



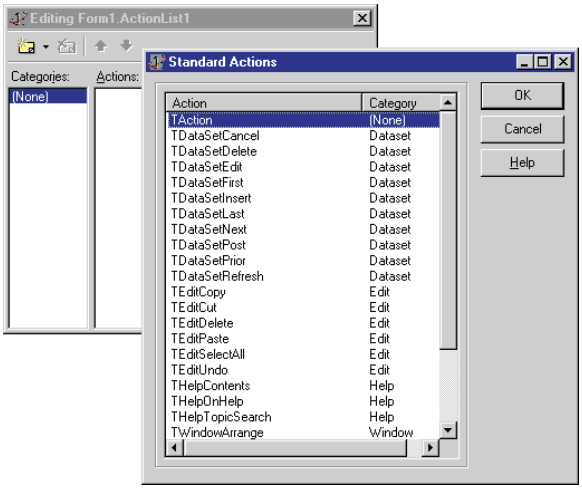
➤ *Figure 5:*
The application with actions.

ActionApp2.dpr is another project, much like ActionApp.dpr except that a `TBitBtn` is used instead of a `TButton`. Also, it employs a popup menu for the listbox, a main menu and a toolbar with a tool button. You will not be surprised to learn that each of these controls is hooked up to the action to prove a point. The action's properties propagate to all the action clients (see Figure 5) and when the action is disabled, all the action clients instantly disable.

Also, these other controls use more of the action's properties. For example, when a `TMenuItem` in the Menu Designer is connected to the action, the `ShortCut` property is copied along with `Checked` and `ImageIndex`. To make sure the menu item's `ImageIndex` had something to index into, both the popup menu and main menu were initially connected to the same image list as the action list.

In the case of a `TToolButton`, the action's `Checked` property value is copied to the `Down` property, but most of the others go through to the correspondingly named property. Tool buttons also have an `ImageIndex` property so the toolbar was also connected to the image list.

One important point about a tool button is worth making. Having connected the toolbar to the image list, the first tool button added to the toolbar will automatically display the action's image. This is just coincidental. The first tool button gets an `ImageIndex` of `0` automatically, the second gets an `ImageIndex` of `1` and so on. The tool button will still need to be connected to the action like any other action client.



➤ *Figure 6: The standard action choice dialog.*

You should be able to see the main point of action components now. Being automatically validated, they give a smooth, reactive user interface with consistency amongst controls that invoke the same functionality.

Standard Actions

The actions that we have been manually setting up are sometimes called *custom actions*, since the Delphi developer customises their behaviour and properties. Delphi also comes with a number of predefined actions, with built-in behaviour and property values, which are referred to as *standard actions*: so named since their behaviour and attributes are built in as standard.

These standard actions provide commonly useful behaviour, such as clipboard interaction, MDI window commands and help commands. Table 1 shows the standard actions that are supplied with Delphi 4 and 5.

You create standard actions from the Action List Editor. However, instead of pressing the yellow button, you should drop down the arrow next to it and choose `New Standard Action...` from the drop down menu (or press `Ctrl+Insert`). This takes you to a dialog that lists all the available standard actions (shown in Figure 6).

Assuming you have an image list associated with your action list, when a standard action is created it will add its associated image into your image list and set its `ImageIndex` property to the position of it, assuming it has one. It also sets up its other property values to predefined values (see Figure 7).

Some of these standard actions have an additional property that can be used to specify a dedicated target control. For example, all the dataset actions have a published `DataSource` property that appears on the Object Inspector. You can optionally use this property to connect the actions to one specific data source component but, again, this is not required. If you leave the property blank, the magic of action handling will allow the action to find the first data source on the active form at runtime.

Similarly, all the standard edit actions have a public `Edit` property and the standard window actions have a `Form` public property. You can therefore programmatically tie an edit action to a fixed edit control, or a window action to one specific form. But, if you do not, the edit action will act on the active edit control (if there is one) and the window action will act on the active MDI form (if there is one).

Classes Involved With Actions

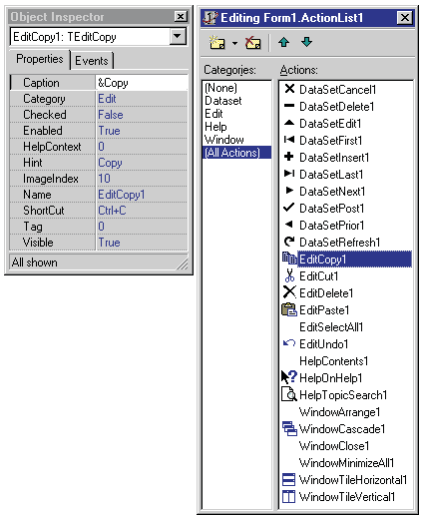
There are a lot of classes associated with actions and so, rather than simply listing them out, I will try and give an overview that encompasses them all. If you have no desire to know more about the internal workings of actions, or how to make reusable standard actions, you should perhaps skip the rest of this article. It is dirty-hand territory from here on in.

➤ *Facing page, Table 1: Standard actions available in Delphi.*

Action Classes

The action class hierarchy starts with `TBasicAction` (see Figure 8 and Listing 4), which can be used in conjunction with an action client that is neither a menu nor a control. `TContainedAction` adds support to allow an action to appear in

➤ *Figure 7: Instances of each of Delphi 5's standard actions.*



➤ *Listing 4: The TBasicAction base action class.*

```
TBasicAction = class(TComponent)
private
  FOnChange: TNotifyEvent;
  FOnExecute: TNotifyEvent;
  FOnUpdate: TNotifyEvent;
protected
  FClients: TList;
  procedure Change; virtual;
  procedure SetOnExecute(Value: TNotifyEvent); virtual;
  property OnChange: TNotifyEvent read FOnChange write FOnChange;
public
  constructor Create(AOwner: TComponent); override;
  destructor Destroy; override;
  function HandlesTarget(Target: TObject): Boolean; virtual;
  procedure UpdateTarget(Target: TObject); virtual;
  procedure ExecuteTarget(Target: TObject); virtual;
  function Execute: Boolean; dynamic;
  procedure RegisterChanges(Value: TBasicActionLink);
  procedure UnRegisterChanges(Value: TBasicActionLink);
  function Update: Boolean; virtual;
  property OnExecute: TNotifyEvent read FOnExecute write SetOnExecute;
  property OnUpdate: TNotifyEvent read FOnUpdate write FOnUpdate;
end;
```

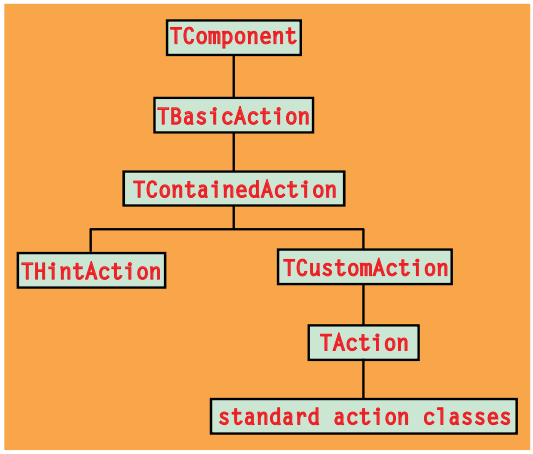
Standard Action Class Name	Introduced In	Defined In	Category	Purpose
TEditCut	Delphi 4	StdActns.pas	Edit	Cuts highlighted text from the target to the Clipboard
TEditCopy	Delphi 4	StdActns.pas	Edit	Copy highlighted text to the Clipboard
TEditPaste	Delphi 4	StdActns.pas	Edit	Pastes text from the Clipboard to the target and ensures that the Clipboard is enabled for the text format
TEditSelectAll	Delphi 5	StdActns.pas	Edit	Selects all the text in the target edit control
TEditUndo	Delphi 5	StdActns.pas	Edit	Undoes the last change made to the target edit control
TEditDelete	Delphi 5	StdActns.pas	Edit	Deletes the highlighted text
TWindowClose	Delphi 4	StdActns.pas	Window	Closes the active MDI child form
TWindowCascade	Delphi 4	StdActns.pas	Window	Cascades the MDI child forms
TWindowTileHorizontal	Delphi 4	StdActns.pas	Window	Arranges MDI child forms so that they are all the same size, tiled horizontally
TWindowTileVertical	Delphi 4	StdActns.pas	Window	Arranges MDI child forms so that they are all the same size, tiled vertically
TWindowMinimizeAll	Delphi 4	StdActns.pas	Window	Minimises all of the MDI child forms
TWindowArrange	Delphi 4	StdActns.pas	Window	Arranges the icons of minimised MDI child forms
THelpContents	Delphi 5	StdActns.pas	Help	Brings up the Help Topics dialog on the tab (Contents, Index or Find) that was last used
THelpTopicSearch	Delphi 5	StdActns.pas	Help	Brings up the Help Topics dialog on the Index tab
THelpOnHelp	Delphi 5	StdActns.pas	Help	Brings up the Microsoft help file on how to use Help
TDataSetFirst	Delphi 4	DBActns.pas	Dataset	Sets the current record to the first record in the dataset
TDataSetPrior	Delphi 4	DBActns.pas	Dataset	Sets the current record to the previous record
TDataSetNext	Delphi 4	DBActns.pas	Dataset	Sets the current record to the next record
TDataSetLast	Delphi 4	DBActns.pas	Dataset	Sets the current record to the last record in the dataset
TDataSetInsert	Delphi 4	DBActns.pas	Dataset	Inserts a new record before the current record, and sets the dataset into dsInsert state so it can be modified
TDataSetDelete	Delphi 4	DBActns.pas	Dataset	Deletes the current record and makes the next record (if there is one, otherwise the previous record) the current record
TDataSetEdit	Delphi 4	DBActns.pas	Dataset	Puts the dataset into dsEdit state so that the current record can be modified
TDataSetPost	Delphi 4	DBActns.pas	Dataset	Writes changes in the current record to the dataset
TDataSetCancel	Delphi 4	DBActns.pas	Dataset	Cancels the edits to the current record, restores the record display to its condition prior to editing, and turns off dsInsert or dsEdit states if they are active
TDataSetRefresh	Delphi 4	DBActns.pas	Dataset	Refreshes the buffered data in the associated dataset by calling its Refresh method
THintAction	Delphi 4	StdActns.pas	None	Undocumented in Delphi 4 and 5, though see the Component Hints On Status Bars entry in this month's Delphi Clinic).

an action list. It also adds the `Category` property to allow actions to be categorised. `TCustomAction` adds the UI properties that can be propagated to action clients such as menus and controls, although it has no published properties. `TAction` publishes all of the interesting properties of `TCustomAction`.

You will notice that `THintAction` is also sitting in the hierarchy (and was listed in Table 1). This class is not documented (at least not in Delphi 4 or 5) as it is intended for internal VCL use. However, we can use it for our own purposes as is explained in the *Delphi Clinic* in this issue.

Action Link Classes

Whilst apparently changing the subject, but not really, I will talk briefly about data-aware controls. Data-aware controls and data source components appear to be directly connected through the data aware controls' `DataSource` property. However, this is not actually the case. Instead, data link objects are employed in any data-aware control that implements a `DataSource` property to act as the liaison officer, represent the link to the dataset and to respond to data events. Similarly, action clients that implement an `Action` property use action link objects to connect action components to their properties (such as `Caption`, `Hint` and `ShortCut`).



➤ *Figure 8: The action class hierarchy.*

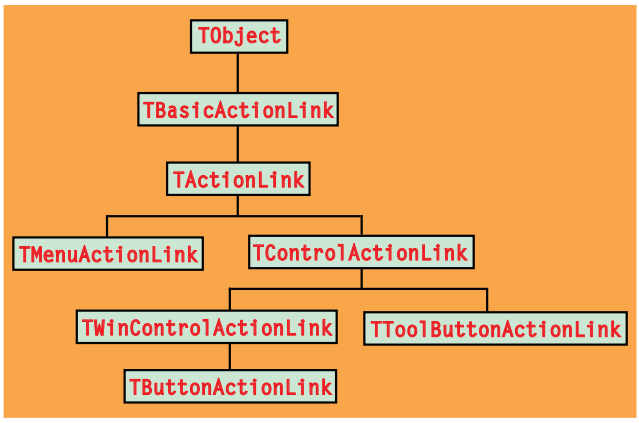
Action links exist as various classes in a mini-hierarchy (see Figure 9) with `TBasicActionLink` at the root (see Listing 5). This class takes the client object as a constructor parameter (although it does not store it) and the related action is available as the `Action` property. It sets up the basic structure of a connection between an action and a client. It defines virtual `Execute` and `Update` methods which call the associated action's `Execute` and `Update` methods. If the action has an `OnExecute` event handler, `Execute` returns `True` and if it has an `OnUpdate` handler, `Update` returns `True`. It also has an `OnChange` event triggered when the properties of the action change.

`TActionLink` adds in basic support for managing the connection between an action's properties and the action client properties (see Listing 6). It has elementary support for `Caption`, `Checked`, `Enabled`, `HelpContext`, `Hint`, `ImageIndex`, `ShortCut` and `Visible`. The `IsXXXXLinked` methods all return `True` if `Action` has been assigned a `TCustomAction` or descendant, whilst the `SetXXXX` methods do nothing. A `TActionLink` can be used as a base class for an action link that can be used when the action client is neither a control nor a menu (which are catered by descendant action link classes).

`TMenuActionLink` adds specific support for menu item clients by overriding `AssignClient` (which stores the client in a private data field) and `IsOnExecuteLinked` from Listing 5, as well as all the virtual methods in Listing 6. This allows actions to map their UI properties to equivalent menu item properties. `TControlActionLink` does a similar job for generic controls, although it only deals with `Caption`, `Enabled`, `Hint`, `Visible` and `OnExecute` (which maps to the client's `OnClick` event).

The other action link classes add support for various other specific properties of their indented client controls. For example `TToolButtonActionLink` works with tool buttons, adding a link between the action's `Checked` property and the tool button's `Down` property (note the difference in name).

Being aware of action links and how they fit in is generally useful, however getting down to the nitty-gritty of how they operate is only important if you wish to write interesting new components with



➤ *Figure 9: The action link class hierarchy.*

➤ *Listing 5: The TBasicActionLink base action link class.*

```
TBasicActionLink = class(TObject)
private
  FOnChange: TNotifyEvent;
protected
  FAction: TBasicAction;
  procedure AssignClient(AClient: TObject); virtual;
  procedure Change; virtual;
  function IsOnExecuteLinked: Boolean; virtual;
  procedure SetAction(Value: TBasicAction); virtual;
  procedure SetOnExecute(Value: TNotifyEvent); virtual;
public
  constructor Create(AClient: TObject); virtual;
  destructor Destroy; override;
  function Execute: Boolean; virtual;
  function Update: Boolean; virtual;
  property Action: TBasicAction read FAction write SetAction;
  property OnChange: TNotifyEvent read FOnChange write FOnChange;
end;
```

new properties which you want controlled by actions. Given that we will not be covering that subject in this article, it is safe to leave action links alone now.

How The Action Architecture Works

Now that we have seen the basic use of actions and had an overview of the classes involved, let's look in more detail at how they work inside a Delphi application. On our travels you will see that the Borland developers have provided many points that can be used to hook into action functionality. The execution path of actions presented here will be quite detailed, to give a full understanding of what goes on.

`Action` is defined as a public property in `TControl` (it is published by a number of descendant classes) and a published property in `TMenuItem`. When you assign an action component to an action client's `Action` property the following sequence of events occurs.

In the case of a control, `csActionClient` is added to its `ControlStyle` set property. Then, all action clients check if their protected `ActionLink` property refers to an action link object. If not, it creates an action link using the class reference returned by the protected dynamic `GetActionLinkClass` method. This returns an appropriate action link class of which an instance is created.

The action link is given the action object and its `OnChange` event is handled by an action client method that copies the key action properties to the action client properties. Since the action has just been set, this routine (a protected dynamic procedure called `ActionChange`) is triggered to get the current action properties copied across. At this point, the client has got the action properties and an appropriate action link object.

How Actions Are Invoked

Now we need to see what happens when an action is invoked. Remember this can happen by an action client invoking it, such as a button being clicked, or by any piece of code calling the action's `Execute` method. It can also happen by the user pressing the shortcut key associated with an action, which need not be connected to any action client. By the end of this section we should be able to see how each of these possibilities works.

We start by taking the case of an action client invoking the action, using an example of a button hooked up to an action. When you look at a button set up as an action client you can see the Object Inspector showing the `OnClick` event connected to the action's `OnExecute` event handler. You might therefore understandably think that clicking the button will simply call the action's `OnExecute` handler. But it is not as simple as that. Oh no.

Instead, assuming that the `OnClick` event has not been changed, the action link's virtual `Execute` method is called (see Listing 7). The default implementation of this in `TBasicActionLink` (which is not overridden in any of the descendant classes) calls the action's dynamic `Execute` method. So, at this point, the action client invoking the action now looks just like the same code explicitly invoking an action by calling the `Execute` method.

The base action class implementation of `Execute` tries to call the action's `OnExecute` event handler. Both `Execute` methods return `True` if the handler exists and `False` if not. However, `TContainedAction` overrides `Execute` to perform more interesting logic. Since contained actions can be managed by action lists, they broaden the potential for response to an action by a wide margin.

Firstly, the action defers to its action list (if it has one) by calling its `ExecuteAction` method and passing itself as a parameter. The action list's `ExecuteAction` method is implemented in order to call its `OnExecuteAction` event handler if it is present. If the event handler sets its `Handled` parameter to `True`, the story ends here. Otherwise, it goes further. Notice that all of the actions in an action list will trigger the action list's `OnExecuteAction` event handler and so some generic handling or action tracking can be implemented there if it is needed.

Next, the `Application` object's `OnExecuteAction` event is triggered

```
TActionLink = class(TBasicActionLink)
protected
  function IsCaptionLinked: Boolean; virtual;
  function IsCheckedLinked: Boolean; virtual;
  function IsEnabledLinked: Boolean; virtual;
  function IsHelpContextLinked: Boolean; virtual;
  function IsHintLinked: Boolean; virtual;
  function IsImageIndexLinked: Boolean; virtual;
  function IsShortCutLinked: Boolean; virtual;
  function IsVisibleLinked: Boolean; virtual;
  procedure SetCaption(const Value: string); virtual;
  procedure SetChecked(Value: Boolean); virtual;
  procedure SetEnabled(Value: Boolean); virtual;
  procedure SetHelpContext(Value: THelpContext); virtual;
  procedure SetHint(const Value: string); virtual;
  procedure SetImageIndex(Value: Integer); virtual;
  procedure SetShortCut(Value: TShortCut); virtual;
  procedure SetVisible(Value: Boolean); virtual;
end;
```

➤ *Listing 6: The TActionLink class.*

```
procedure TControl.Click;
begin
  { Call OnClick if assigned and not equal to associated action's OnExecute.
    If associated action's OnExecute assigned then call it, otherwise, call
    OnClick. }
  if Assigned(FOnClick) and (Action <> nil) and
    (@FOnClick <> @Action.OnExecute) then
    FOnClick(Self)
  else if not (csDesigning in ComponentState) and (ActionLink <> nil) then
    ActionLink.Execute
  else if Assigned(FOnClick) then
    FOnClick(Self);
end;
```

➤ *Listing 7: The implementation of TControl.Click.*

if present. Similarly, if the Handled parameter is set to True, processing ends there, otherwise it carries on. Note that the Application object's OnExecuteAction event will be triggered for all actions in the application that are not handled by their corresponding action list, providing an option for completely generic action handling, or alternatively a way to intercept actions not handled by their action list.

If the action still has momentum, it at last tries to execute its own OnExecute event handler. If no handler was set up, which will be the case for standard actions, it goes still further. It packages itself up in a CM_ACTIONEXECUTE message which gets sent to the Application object's window as a cry for help in trying to find a target to execute against. The Application object's message handler calls its DispatchAction method, which tries to locate a target on the active form and, failing that, the main form.

It does this by sending the same message to each form in turn, assuming it exists. The form then verifies that it is visible and, if so, tries to find a target control. Firstly, it checks whether the active control is a suitable target by passing the action object to the control's ExecuteAction method. If not, the form itself is tested to see if it might be a target, passing the action to its own ExecuteAction method. If there is still no joy it calls ExecuteAction for every visible control on the form, stopping if it finds a match.

The implementation of a component's ExecuteAction method typically involves the component passing itself to the action's HandlesTarget method. If this returns True, we have a suitable target and so the target is passed to the action's ExecuteTarget method.

This way, an action target can be found for an action without the target knowing anything about the action in advance. However, ExecuteAction can be overridden to allow any control to pick up specific actions of interest, if needed, without the action knowing about the target. It works both ways.

If the main form does not successfully handle the message then the final step is reached. If the action is a TCustomAction (or inherited from that class), is currently enabled, has no OnExecute handler and its DisableIfNoHandler property is True, then the action is disabled. DisableIfNoHandler is a public property defined by TCustomAction which defaults to True.

TCustomAction also overrides this Execute behaviour from TContainedAction to call the virtual Update method to update the action's state before setting off on the possibly lengthy trek to execute the action. This ensures the action's state is up-to-date, based upon the immediately current state of the application before being executed.

The case we have not looked at yet is where an action's shortcut key is pressed, causing the action to be invoked, regardless of whether an action client has been set up or not. Whenever a keystroke is pressed and is not handled by the active control or a suitable popup menu item, it is passed to the underlying form's IsShortCut method.

The form tries to handle the keystroke through its OnShortCut event or, failing that, through its main menu. If nothing wants it, all action lists owned by the form are checked for a matching shortcut. The action list checks each of its actions and if a match is found, the action's Execute method is called.

If no suitable action is found on the current form, a CM_APPKEYDOWN message is sent to the Application object, which calls its own IsShortCut method. This tries to handle the keystroke in its own OnShortCut event and if that fails it calls the main form's IsShortCut method.

This way a shortcut key can be picked up by an action on the active form or the main form, assuming a menu item or OnShortCut event does not handle it first.

As you can see, the VCL goes to a lot of trouble to service actions if they do not have an OnExecute event or even an action client, which standard actions tend not to. It is this concerted search effort that enables standard actions to work without necessarily being connected to an action client. They can typically operate on the active control (or some other suitable control) on the active form thanks to the VCL's inbuilt target-searching logic.

How Actions Are Updated

As I mentioned, TCustomAction updates an action just before trying to execute it. However, they also get updated at another time. When an application has processed all of its pending messages it transitions into an idle state. Windows wakes it up when another message arrives. Just before going idle the Application object's Idle method calls its OnIdle event handler and then calls the DoActionIdle method.

DoActionIdle loops through all enabled forms on-screen calling UpdateActions, which calls the virtual InitiateAction method for itself, all top level, visible menu items and then all visible controls with csActionClient in ControlStyle (in other words all action clients). InitiateAction calls the Update method of the action link, if there is one which calls the action's Update method.

The TBasicAction implementation of Update either calls OnUpdate if it exists and returns True, otherwise it returns False. TContainedAction overrides Update to do much the same sort of thing as with Execute. It checks to see if the action list or Application object wishes to deal with updating the action in their OnUpdateAction events. Then it tries its own OnUpdate event handler. If there is no handler it asks the Application object to help find a target control to update itself against.

The action is passed to each possible target's UpdateAction method which calls HandlesTarget. If the action claims to handle the target then the action's UpdateTarget method is called.

This all means that every time a user's input (key presses, mouse clicks, and so on) have

been serviced and the program goes idle waiting for the next message, all actions connected to action clients are updated. Consequently, this means that the action clients always have an up-to-date representation of the action properties. Additionally, all actions (regardless of whether they are connected to clients or not) are updated just before they execute.

Because the CPU is so fast, the application will go idle between each key press and mouse click, meaning that actions are updated very regularly. The implication of this is that you must ensure your action update code is not time-intensive, to avoid having a sluggish application.

How Standard Actions Are Made

We have seen how to make custom actions and how to use standard actions, so now we turn our attention to making new standard actions of our own. The general idea is fairly straightforward, although there are a few twists here and there.

The plan is to inherit a class from `TAction` and override three methods. `HandlesTarget` decides whether we handle a given target control. `UpdateTarget` should check appropriate criteria and update the action properties if needed. `ExecuteTarget` contains the code represented by the standard action. These methods have all been mentioned before and are all defined as virtual in `TBasicAction` where they do nothing except `HandlesTarget`, which returns `False`, indicating nothing is handled.

If needed, the action can define a property to link it to a specific target component. If this is done you must be careful to hook into the standard notification mechanism so you are informed if the linked component is destroyed.

The standard actions we will develop will enable the user to scroll through the tabs on a tab control or the pages on a page control (both of which inherit from `TCustomTabControl`). A property will be made available to hook either type of control to the action, although this will not be strictly necessary, as the action knows it can handle any tab control or page control thrown at it.

We will make two actions, `TNextTab` and `TPriorTab`. These will have dedicated shortcuts of `Ctrl+Tab` and `Ctrl+Shift+Tab` as used by the IDE. `TNextTab` will select the next available tab or page and will go back to the first one if the last one was selected. Similarly, `TPriorTab` will select the previous tab, going to the last one if the first one is already selected.

There will be a certain amount of commonality between these actions, so a base class will be defined first called `TTabAction`. As Listing 8 shows, it defines a published `TabControl` property that allows a tab control or page control to be assigned, storing the object reference in `FTabControl`. If a new control is assigned, it is told to make sure we are notified if it gets destroyed.

`HandlesTarget` is defined to work like other standard actions. If we have a `TabControl` property value, we only claim to handle the offered target if it is the same control. If `TabControl` is blank, we will handle any tab control or page control. The code is being picky about control types as `TTabControl` and `TPageControl` use different properties for selecting tabs and returning the number of tabs. `TTabControl` uses `TabIndex` and `Tabs.Count`, whilst `TPageControl` uses `ActivePage` and `PageCount`. Other `TCustomTabControl` descendants could introduce yet more indexing options which the code would not know how to deal with.

`UpdateTarget` disables the target if there is only one available tab. Notice that the tab count is also given by different properties for tab controls and page controls.

Now we can implement the real actions, as per Listing 9. You can see the tab index or page index being modified as appropriate in

➤ *Listing 8: The common base class for the standard actions.*

```
type
  TTabAction = class(TAction)
  private
    FTabControl: TCustomTabControl;
  protected
    procedure SetTabControl(Value: TCustomTabControl);
    procedure Notification(AComponent: TComponent;
      Operation: TOperation); override;
  public
    function HandlesTarget(Target: TObject): Boolean;
      override;
    procedure UpdateTarget(Target: TObject); override;
  published
    property TabControl: TCustomTabControl
      read FTabControl write SetTabControl;
  end;
procedure TTabAction.Notification(AComponent: TComponent;
  Operation: TOperation);
begin
  //Set target control property to nil if target is destroyed
  if (AComponent = FTabControl) and (Operation = opRemove)
    then FTabControl := nil
end;
procedure TTabAction.SetTabControl(Value: TCustomTabControl);
begin
  if Value <> FTabControl then begin
    FTabControl := Value;
    //If we have a target control, request notification
    //so we will be told if it is destroyed
    if Assigned(Value) then
      Value.FreeNotification(Self)
  end;
end;
function TTabAction.HandlesTarget(Target: TObject): Boolean;
begin
  //If we have a specific target and it matches
  //the passed control, we will accept it
  if Target = FTabControl then
    Result := True
  else
    //If we have no specific target, but the passed
    //control is of an acceptable type, then we accept it
    if (FTabControl = nil) and ((Target is TTabControl) or
      (Target is TPageControl)) then
      Result := True
    else
      Result := False;
end;
procedure TTabAction.UpdateTarget(Target: TObject);
begin
  //Make sure tab control has more than one tab
  if Target is TTabControl then
    Enabled := TTabControl(Target).Tabs.Count > 1
  else
    //Make sure page control has more than one page
    if Target is TPageControl then
      Enabled := TPageControl(Target).PageCount > 1
    else
      //No other TCustomTabControl derivative is understood
      Enabled := False;
end;
```

```
type
  TPriorTabAction = class(TTabAction)
  public
    procedure ExecuteTarget(Target: TObject); override;
  end;
  TNextTabAction = class(TTabAction)
  public
    procedure ExecuteTarget(Target: TObject); override;
  end;
procedure TPriorTabAction.ExecuteTarget(Target: TObject);
begin
  if Target is TTabControl then
    with TTabControl(Target) do
      TabIndex := (TabIndex - 1 + Tabs.Count) mod Tabs.Count
  else
    if Target is TPageControl then
      with TPageControl(Target) do
        ActivePage := Pages[(ActivePage.TabIndex - 1 +
          PageCount) mod PageCount]
end;
procedure TNextTabAction.ExecuteTarget(Target: TObject);
begin
  if Target is TTabControl then
    with TTabControl(Target) do
      TabIndex := (TabIndex + 1) mod Tabs.Count
  else
    if Target is TPageControl then
      with TPageControl(Target) do
        ActivePage :=
          Pages[(ActivePage.TabIndex + 1) mod PageCount]
end;
```

➤ *Listing 9: New standard actions.*

```
procedure Register;
begin
  RegisterActions('Tab', [TNextTabAction, TPriorTabAction], TTabActionsModule)
end;
```

➤ *Listing 10: Registering the actions.*

the `ExecuteTarget` methods. This code can be found in the file TabActns.pas, which has been added into the TabActions40.dpk and TabActions50.dpk runtime packages on this month's disk, for use with Delphi 4 and 5 respectively. Don't forget to place the compiled package (the .BPL file) in a directory on the path to allow Delphi to see it.

Initialising Standard Actions

Apart from initialising their properties, the actions are now ready to be registered. Typically, components perform property initialisation in their constructors. This would work fine for these actions as well except for the associated image.

You may recall that standard actions can copy their image into the image list associated with your action list (see Figure 7 for a reminder). In order to allow our actions to be just as friendly, we initialise the properties in a different way to normal components.

Registering Standard Actions

To register standard actions in the IDE, you call `RegisterActions`, passing a category, a list of action classes and, optionally, a data module class. The data module is used to pre-initialise the action properties and image. It works like this.

You make a data module, then drop an image list onto it, which you fill with images for your standard actions. Next, you place an action list on the data module and hook it up to the image list. The idea is that you then use the Action List Editor to create instances of your new standard actions, whose properties you can initialise as you like. This data module class is then passed as the third parameter to `RegisterActions`.

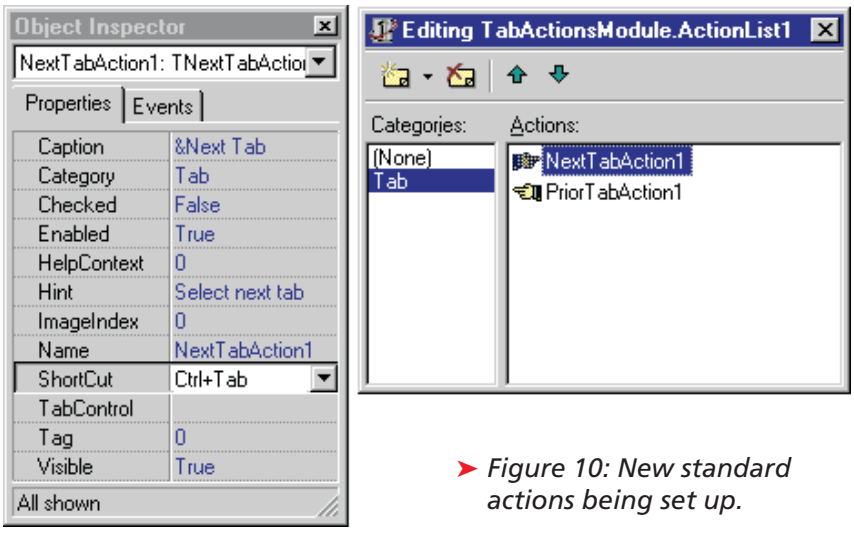
However, if you think about this, in order to get your standard actions created through the Action List Editor they must first be registered so that the IDE knows about them. So what we can do is ignore the data module issue to start with and register the actions on their own (we can worry about the data module later).

This is done with a normal IDE registration routine in a registration unit (TabActnsReg.pas). The routine looks much like the one in Listing 10, but with `nil` passed in place of the data module class. The registration unit is added to a design-time package (file DCLTabActions40.dpk for Delphi 4 or DCLTabActions50.dpk for Delphi 5, both on this month's disk of course) which is compiled and installed.

Standard Actions And Data Modules

At this point the IDE knows about the new standard actions so the data module can now be set up. A suitable data module called `TabActionsModule` is in the unit TabActionsRegModule.pas which should be added to the design-time package. The image list can now be set up to contain appropriate images. The action list can be added and connected to the image list.

The Action List Editor can be used to create an instance of each of our new standard actions, and their properties can be set as required. Figure 10 shows one of the actions fully set up on the data



➤ *Figure 10: New standard actions being set up.*

module, with all its custom properties on the Object Inspector.

Now all that is left is to modify the registration routine to refer to the data module class. Listing 10 shows the final version. One final compile of the design-time package and the job is done. Two new fully-fledged standard actions are available for use.

You are now free to make a new application with a tab control and/or a page control on it, set up an action list with both the new actions in and test them out. You have the option of tying the actions to one of these target controls, but this is not necessary. You also have the option of connecting the action to a specific action client, but similarly this is not necessary (they have a specified shortcut after all).

At runtime pressing `Ctrl+Tab` or `Ctrl+Shift+Tab`, or pressing the action client if there is one, will invoke the corresponding action which will act either on the control connected to the `TabControl` property if there is one. If not, it will act on the focused control or the first suitable control encountered.

Summary

Actions represent a very convenient and manageable way to implement user-driven functionality that can be invoked in a variety of ways. Whilst currently underused by many developers, hopefully as more people become aware of their power and benefit, they will become more commonplace in Delphi application development.

Brian Long is a UK-based freelance consultant and trainer. He spends most of his time running Delphi and C++Builder training courses for his clients, and doing problem-solving work for them. Brian is at brian@blong.com