Data Validation Techniques
In Turbo Vision

by Danny Thorpe

presented at the
Borland Languages Conference
April 1991

Data Validation Techniques in Turbo Vision

Turbo Vision, the new object-oriented event-driven application framework introduced in Borland's Turbo Pascal 6.0, has generated quite a bit of excitement in programming circles. With Turbo Vision, programmers finally have the tools and application framework necessary to create programs that can truly benefit from the OOP methodology. Programmers benefit from the variety of objects in Turbo Vision that can be molded and extended into a multitude of application-specific dialog boxes, windows, and other views.

Turbo Vision is a multi-faceted environment. At a superficial level, it is a library of interrelated objects for managing the screen display. At another level, though, Turbo Vision is also an application framework which defines a particular model of how a program should work, both internally and visually. It is this model and its framework embodiment that provides the glue which ties the objects in the library together into a functional society of objects. A large portion of the increased programmer productivity that can be realized with Turbo Vision is due to this model as well: the application framework you inherit takes care of the actual mechanics of the program, so the programmer has just to plug in modules and create cross-connections between objects.

While many programmers appear to accept the code changes necessary to move to Turbo Vision, some are reluctant to also embrace the programming model that is an important part of Turbo Vision. Without this model, one can wind up using Turbo Vision objects in ways that work against the grain of the programming model, creating unnecessary complexity and frustration for the programmer.

A large portion of any program involves feeding in data for the program to operate on. Traditional techniques for interactive data entry are particularly difficult for programmers to give up when learning Turbo Vision. This paper will discuss some of the philosophy behind Turbo Vision, the mechanics of various internal operations in Turbo Vision, and why some of the old data entry techniques won't work anymore. Alternate approaches to data entry and validation more in line with Turbo Vision model will be introduced and demonstrated.

The Turbo Vision Model

The Turbo Vision user interface is based upon the IBM System Application Architecture, Common User Access specification. CUA provides guidelines that determined the behavior of the mouse, input lines, push-buttons, and the choice of keys for particular actions, such as Esc to cancel a dialog and Enter to select the default button of the dialog.

A driving force in the Turbo Vision programming model is the idea that the user of the program should be in control of the program at all times. Let the user do what he wants to do, within reason. This doesn't sound very remarkable at first, but when you consider how few programs on the market allow the user this freedom, you begin to realize that an unnecessary evil has been lurking in computer software: too many restrictions are imposed upon the user by the program/programmer, to the point that the software is in control of the user.

Of course, a program needs structure in order to organize its information into a meaningful form. The user needs the information to be structured, too, in order to understand the material. The difficulty comes in finding the fine line between the amount of structure needed by the user and the amount of structure that begins to stifle the user with unnecessary conditions and requirements. Programs frequently wind up on the stifling end of the scale simply because they're easier to write and manage. With traditional programming tools it is difficult to construct a well-balanced, flexible program that considers all contingencies the user may wish to embark upon.

With Turbo Vision, you inherit a well-balanced, flexible program architecture and then fill in the details of what your application is to do. At the lowest levels in the object hierarchy, Turbo Vision views have the ability to be manipulated by the user in a multitude of ways. But the programmer doesn't have to bend over backwards in Turbo Vision to accomplish this user freedom. All the programmer has to do is work within the Turbo Vision model to extend the behavior of view objects that already know how to interact with the user.

The Turbo Vision model provides an environment based on autonomy and independence. The careful integration of OOP and event driven programming provides the programmer with a rich set of tools and opens doors to programming solutions nearly unattainable by traditional methods. Working in this model actually provides the programmer more flexibility in expression and more opportunity to bring sophisticated applications to completion than traditional programming tools can offer.

Traditional Validation Techniques

As-You-Type Validation

One method of ensuring an input field contains "clean" data is to monitor each keystroke the user types. Keys can be checked by position within the input field if the validation is controlled by a template. Formatting telephone number fields with a template like (999) 999-9999, where '9' means 'accept any number in this position' is a common use of as-you-type validation. If a non-numeric key is pressed, the key monitor can beep or provide some other non-intrusive indicator of an error. The other characters in the template can be treated just as place-holders, so the user doesn't have to enter the parentheses for every phone number, for example.

As-you-type validation is good for position-sensitive formatting of data, such as telephone numbers or dates, and for character conversions, such as converting all characters entered in a field to uppercase as they are typed. As-you-type is difficult to use when numbers or whole words need to be validated before being accepted, because you don't really know what the user intends until the user is finished typing, backspacing, and correcting misspellings.

Templates are easy to implement in Turbo Vision: make a descendent of a TInputLine object and extend the HandleEvent method to test each keystroke received by that view. Acceptable keystrokes can be passed to the inherited HandleEvent for default processing, and unacceptable keys can be discarded with a beep and/or non-intrusive error message.

As-You-Leave Validation

Another opportunity to test the data is when the user moves the cursor out of the field, presumably to the next field on the screen. This method assumes that when the user hits the Tab key or Enter key to move to the next field, the user is finished keying this field, and so it is fair game for validation. Validation is performed before the program moves focus to the next field.

If the contents of the field are unacceptable, it is common for an error message to be displayed, and the user is put back in the bad field to reenter it. This is where a poorly designed data entry system can be really mean to the user - what if the user decides to cancel the whole data entry operation? It would be the job of every field to watch for a particular "cancel" key, such as F1 or Esc, and shut down the data entry mode when such a keystroke occurs.

Data can be tested as the user leaves a field in Turbo Vision. However, the traditional response of forcing the user back to a field after the user has indicated he wants to go elsewhere conflicts with the Turbo Vision model. Consequently, this behavior is difficult to implement in Turbo Vision, mainly because it requires different assumptions about the user and environment than Turbo Vision assumes. A detailed explanation of the focus change mechanism is covered in the next section.

As-you-leave validation assumes that when the user leaves the field, the field is ready to be validated. This is a safe assumption in a single-window keyboard-only program. But Turbo Vision's easy access to multi-windowed text and mouse support invalidates this assumption, because the user could be leaving the field to go to an entirely different window, such as a text editor or calculator.

Assume for a moment that a programmer has implemented an As-you-leave, force-back-if-invalid validation scheme for Turbo Vision views. Picture a dialog box with a few data entry lines, a close-icon on the frame, an Ok button, and a Cancel button. The user enters invalid data in one of the input fields and presses the tab key to move to the next input field. The validation routines kick in when the current field is about to lose focus, detect the invalid entry, and prevent the user from moving to the next input field until valid data is entered. A mild frustration for the user, perhaps, but a liveable situation.

Let's say the user enters invalid data in the input field, then clicks the mouse on the Cancel button in the dialog. This will cause a focus change from the current input field to the dialog button. Since controls in a dialog usually operate completely oblivious of each other, the input field has no way of knowing where the focus is going to when it loses focus. In this scenario, the validation routines would still force the user back to the input field until the field contained valid data. What's the point of having a Cancel button if you can't use it when you need it most? This is an unacceptable situation.

For the sake of argument, let's say the programmer recodes all the input field objects with throwback validation to recognize a focus change to a special Cancel button and allows the cancel to take place. The user is appeased for the moment, but what happens when the user demands a new button on that dialog? More special casing, more hidden interdependencies?

The programmer has committed one of the worst sins of object-oriented programming - special casing in general purpose objects - which quickly defeats many of the advantages of OOP: independence, flexibility, maintainability. As soon as the user demands a new button on that dialog, the programmer must recode all the input field objects related to that throwback validation scheme, expending more time and energy on an old problem, possibly destabilizing the entire application, and never really solving the problem to begin with. All this, just to add a simple button.

As-you-leave validation is not improper in Turbo Vision, if it is done non-intrusively. A simple As-you-leave validation response to bad data is to simply change the color of the field. The user will see the possible error and can address the problem as he sees fit.

When-You're-Done Validation

A third opportunity to test the data is when the screen or record is complete and the user indicates he is finished entering or editing the data. All forms of validation can be performed at this stage because the data is complete and the user has indicated the data is complete. No assumptions to tippy-toe around here. If one of the fields contains unacceptable data, the user can be returned to the bad field with an error message. The problem of providing the user an escape still exists.

When-you're-done validation is Turbo Vision's default validation mode. When the user exits a dialog, the Valid method of every view in the dialog is called. If any Valid method returns False, then the dialog will not close. If the user cancels the dialog, no validation calls are made and the data in the dialog is discarded.

This data validation method is the most flexible in Turbo Vision. One can extend the behavior of TDialog, for example, to perform data validation and respond to errors in a variety of ways. As detailed later in this paper, a dialog can be made to find which view is invalid and return the user to that view for corrections. And by validating the data when the user indicates he's ready, we avoid all the problems associated with As-you-leave validation in a multi-windowed environment.

Focus Change Internals

When a user hits the Tab key to move from one dialog control to another, a rapid sequence of events and object method calls occurs to transfer the focus of the dialog from one view to another. Programmers tend to attack this focus change sequence when attempting to force Turbo Vision to return focus to a bad field in an As-you-leave data validation. This section describes the sequence of events in a focus change and why it should not be aborted in mid-stream.

Turbo Vision terms will be flying fast and furious in a moment. If you are not familiar with the terms view, group, focused, and selected, take a minute to review the terminology appendix at the end of this paper.

Press the Tab key.

Let's say the user is looking at a dialog which has two input lines. When the user presses the Tab key, an evKeydown event is generated, which is sent to the dialog's HandleEvent method. The Tab key event is passed back to ancestral handleevents and eventually gets processed by TWindow.HandleEvent. For a Tab key, TWindow.HandleEvent calls TGroup.SelectNext. TGroup.SelectNext scans through the group's (dialog's) list of views for the next selectable view. In this example, the next selectable view is the second input line. The second input line's Select method is called, which by inheritance works out to be TView.Select.

TView.Select calls TGroup.SetCurrent to make itself (the second input line) the current view.

TGroup.SetCurrent

The TGroup object contains a private method called SetCurrent that performs the dirty work of changing the focus from one view to another. It is a private method because the operations it performs are low-level and are not considered safe to modify.

TGroup.SetCurrent does four things of importance, and all of them involve setting the state flags of the two views transferring focus. First, the currently focused view is de-focused, then de-selected. Next, the new view to receive focus is selected, then focused.

We'll trace what happens when a view changes state in a moment. Right now, it's important to observe that there is no room for arbitration in this rapid fire sequence of defocusing and refocusing. View A is told to de-focus and de-select itself. Then View B is told to select and focus itself. If a programmer were to modify a view's SetState method to refuse to de-focus itself when told to, the group would be corrupted with what would appear to be two focused views, even though only one view can be focused at a time.

The group assumes that state changes (like from focused to not focused) will be successful. This is in sync with the Turbo Vision program model, which assumes that you can always switch between non-modal views in an application. This also greatly simplifies the logic of focus changes and other operations that change a view's state, because we don't have to worry about how to "back up" if a state change failed late in a focus change sequence.

Tracing the SetStates

Let's call the first input line which is losing focus View A, and the input line that's receiving focus View B. TGroup.SetCurrent calls ViewA^.SetState(sfFocused, False). TView.SetState sends a cmReleasedFocus broadcast event to the view's owner (the dialog) via the Message function. The Message function takes an event as parameters and pipes it directly into the destination object's HandleEvent method.

The dialog's HandleEvent receives a cmReleasedFocus broadcast event. By default, TDialog doesn't do anything with the event, and none of the ancestral HandleEvents do anything with it either. This message is mainly to inform whoever might care that the view is releasing focus. When the event has run its course through the dialog's subviews, the Message function will return control to the TView.SetState which called it.

Control returns to TGroup.SetCurrent, which next calls ViewA^.SetState(sfSelected, False). TView.SetState simply clears the sfSelected flag in its State field, but some TView descendents (such as TButton) may redraw themselves to reflect different colors for selected vs non selected states.

Control returns to SetCurrent, which next calls ViewB^.SetState(sfSelected, True). The sfSelected

flag is set in ViewB, which may cause a redraw to reflect colors of the new state.
Control returns to SetCurrent, which last calls ViewB^.SetState(sfFocused, True). TView.SetState sends a cmReceivedFocus broadcast event to its owner-dialog via the Message function. The dialog receives the event in its HandleEvent, but like the cmReleasedFocus broadcast, this broadcast event is mainly for informational purposes and is not used by any of the inherited HandleEvents.
When control again returns to SetCurrent, the group's Current pointer is set to point to ViewB. The focus change sequence is complete.
Summary
Turbo Vision assumes that state changes, particularly selection and focus state changes, will succeed when invoked. This section has stepped through a focus change sequence to illustrate who all the players are behind the scenes and show the roles they play. Attempting to interrupt or short-circuit a focus change will leave the dialog in an unstable, probably unusable state.

Demo Programs

So you don't go away empty handed, here are two very simple demo programs that illustrate how the interplay between certain TView and TGroup methods can be used to perform data validation. These examples are primarily to show where you can plug in validation code - not how to validate your data.

As-You-Leave Validation

The Valid1.pas program on the disk accompanying this paper implements data validation code in a TInputLine descendent that checks the data when the input line loses focus. If the data is unacceptable, the input line is redrawn in red. It will get the user's attention, but will not interfere with what he is doing.

TValidInputLine has three main parts, and all of them are very small: a Valid function to test the data, a SetState procedure to call Valid when the view loses focus, and a GetPalette function to map an error color (flashing white on red) into the standard TInputline color palette when the input line is in an error condition. All these methods are extending inherited ancestor methods and behaviors. TValidInputLine also has a new boolean field, IsValid, to keep track of what the result of the last Valid check was and to determine whether GetPalette returns an error color palette or its standard color palette. As soon as the error color is no longer needed or wanted, IsValid is set to True, which will cause the standard color palette to be used the next time the view is drawn.

TValidInputLine looks like this:

```
type

TValidInputLine = object(TInputLine)

                                                    IsValid:
Boolean;

                                                    constructor
Init(var Bounds: TRect;

AMaxLen:Integer);

                                                    function
GetPalette: PPalette;  virtual;

                                                    procedure
SetState(AState: Word;
```

```
                                                                  Enable:
Boolean);  virtual;

                                                                  function
Valid(Command: Word): Boolean; virtual;
                                                                  end;
constructor TValidInputLine.Init(var Bounds: TRect;

                                                                  AMaxLen:
Integer);
begin
  TInputLine.Init(Bounds, AMaxLen);
                                                                  IsValid :=
Valid(cmOk);
end;
function TValidInputLine.GetPalette: PPalette;
const  AltPalette: String[Length(CInputLine)]

                                                                  = CInputLine;
begin
  AltPalette[1] := #255;
                                                                  if IsValid then
                                                                  GetPalette :=
TInputLine.GetPalette
                                                                  else
```

```
                                                                GetPalette :=
@AltPalette;
end;
procedure TValidInputLine.SetState(AState: Word;
                                                                  Enable:
Boolean);
begin
                                                                if ((AState
and sfFocused) <> 0) and


GetState(sfFocused) then

                                                                Valid(cmOk);

TInputLine.SetState(AState, Enable);
end;
function TValidInputLine.Valid(Command: Word): Boolean;
begin
  if Command <> cmCancel then
                                                                begin

                                                                IsValid :=
(Data^ = 'Hello');

                                                                Valid :=
IsValid;

                                                                Write(#7);
{ beep }

                                                                DrawView;
{ Show new colors }
                                                                end;
end;
```

The Init constructor initializes IsValid by calling the Valid method. Whenever the view loses

focus, the SetState method calls Valid, which checks the string data of the input line against the only valid response - 'Hello'. IsValid is set and DrawView is called to display the new colors. DrawView will use GetPalette in the process of redisplaying the input line, and GetPalette knows to check IsValid to see which palette should be returned.

Compile and run the simple shell program and dialog that is Valid1.pas. You will notice the input line is immediately red, because the Init constructor calls Valid and the string that's in the input line is 'Phone home.' not the 'Hello' that Valid is looking for. When you begin to type, the input line returns to its normal colors. When you Tab to the Ok button, Valid is called and the error color may be redisplayed, depending upon what you typed.

Selecting the Cancel button, pressing Escape, or clicking on the close icon of the dialog will all cancel the dialog with no further validation. Selecting the Ok button or pressing Enter will close the dialog with a cmOk command, and a final round of validation is done before the dialog closes. If the input line is not valid, the dialog will not close. The user must either correct the data or cancel the dialog.

When-You're-Done Validation

Turbo Vision dialogs already verify that all their controls return Valid true before the dialog is allowed to close. Valid2.pas selects the first control that returned Valid false, so the user is placed on the control with bad data. Valid2.pas also displays the control in the red error color.

Valid2.pas uses a similar TValidInputLine object: the only things that are different in Valid2 are that the SetState method was dropped and a HandleEvent method was added, and the DrawView call was removed from the Valid method.

Valid2 adds a new dialog object type, TDemoDialog, which has simply a new Valid method which overrides the inherited TDialog.Valid method. This new method finds the first control whose Valid method returns false, and selects that control to make it the currently focused view.

Compile and run Valid2.pas. Tab around the dialog a bit, then select the Ok button. The dialog beeps as the input line's valid method objects to closing, the input line is painted red, and the input line becomes the focused view. When you begin typing into the input line or Tab to another view, the input line is redisplayed in its normal colors.

End.

Though extremely simple, these two programs should provide a breadboard upon which you can build and test much more elaborate data validation routines of your own. Armed with some additional knowledge of Turbo Vision's internals, your data validation routines can also work with Turbo Vision, instead of against it, for greater longevity of the program and programmer.

Appendix

Terminology

Let's review some Turbo Vision terms. A view is any TView object or any TView descendent object, including TGroup, TWindow, TDialog, TButton, TInputline, etc. The simplest displayable object in Turbo Vision is a TView, and everything visible on the screen is managed by an object which is in some way descended from TView.

Similarly, a group is a TGroup or any descendent of a TGroup, such as a TWindow or TDialog. A group is a view which maintains a list of other views which the group is said to "own." Views are inserted into groups, groups maintain a coordinate system relative to the group's rectangle origin, and groups distribute events to their subviews in an order that's determined by the type of event.

The focused view is the view which receives focused events like keystrokes and command events. There can be only one focused view in an application at any time.

A selected view is the view in a group which the group's Current pointer is pointing to. Each group can have only one selected view, but since the application can contain many groups, there can be many selected views in an application at one time. When a group receives focus, it gives the focus to its selected view. When a view loses focus to a view in a different group, the view retains its selected state in its group but relinquishes its focused state.

A focused view will always also be selected. However, a view can be selected without being focused. Basically, the selected state is a place holder for which view in a particular group should receive focus when focus returns to that group.

Most of the behavior of a dialog is inherited from the TGroup object type, so many of the methods discussed will be described as TGroup methods, not TDialog methods. Understanding what parts of an object are inherited, and therefore shared by all descendents of an ancestor, is important to get the most out of Turbo Vision. When you want a dialog that has a slightly different Execute method, for example, its useful to know that the default Execute method is inherited from TGroup, but when you actually implement the new Execute method, it will be in a TDialog descendent object.

Acknowledgements

Chuck Jazdzewski, for guidance.
Eberhard Waiblinger, for time.
Kurt Diesch, for questions.