BoldSoft

Creating custom Bold aware components

Table of contents

Table of contents 2

Introduction 3

What's covered in this Document 3
Other documentation 3
Document conventions 3

Creating Bold-Aware components 4

The anatomy of a Bold-Aware component 4
Creating a single-value Bold-aware component 4
Creating a TBoldDateTimePicker *4*
Creating the TBoldDateTimePicker class *5*
Adding the helper objects *5*
Making the component display values *9*
Making the component editable *9*
Implementing ***bapExit*** *10*
Implementing ReadOnly on a component *11*
Implementing `Kind=dtkTime` *13*
Making the ocl-editor work *14*

Chapter 1.

Introduction

What's covered in this Document

This document describes creating custom Bold-aware components. The document assumes that the reader is familiar with the basics of Bold, as well as with creation of custom components in Delphi.

Other documentation

The *Bold for Delphi Developer's Guide* describes the basics of Bold. The *Bold for Delphi Developer's Reference* contains a description of the architecture of the Bold-aware components.

Document conventions

The Bold for Delphi manuals follow the same conventions used by Borland International in the Delphi manuals.

Typeface or symbol	Meaning
`Monospace type`	Represents text as it appears on screen or in Object Pascal source code. Also used when designating text you must type.
[ ]	Square brackets in text or syntax listings enclose optional items not to be typed verbatim.
Boldface	When used in text or source code listings, bold type denotes Object Pascal reserved words or compiler options
Italics	Italic type in text represents Object Pascal identifiers such as type or variable names. Italics may also be used for emphasis, particularly when introducing terminology for the first time.
KEYCAPS	Indicates a keyboard key, or keystroke combination. For example "Press ESC" or "Press SHIFT+F1".
■	Indicates the beginning of a step-by-step procedure
➢	Indicates a specific step you should take in a tutorial
✓	Indicates a summary of what you have accomplished in preceding steps

Chapter 2.

Creating Bold-Aware components

This section describes the creation of Bold-aware components. It does so through a number of examples of designing components. The theory behind the operation of the Bold-aware components is covered in the Bold Developers Reference.

The anatomy of a Bold-Aware component

There are two basic kinds of visual components in Delphi. One type actually holds the value to be displayed in properties belonging to the component, such as `TEdit` and `TListBox`. The other type of components simply paint values on the screen, such as a `TCustomGrid`. The Bold-aware architecture caters for both kinds. It does this by using a number of supporting classes. The ideas behind the architecture are described in the *Bold for Delphi Developer's Reference*

Creating a single-value Bold-aware component

The simplest type of custom component is one that displays a single value, such as an edit-box. A component of this type has a `BoldHandle` property, which is set to a handle to the element to be displayed, and a `BoldProperties` property, which specifies how the element should be displayed. This section describes process for creating such a component step-by-step.

Creating a TBoldDateTimePicker

In this section, we will create a bold-aware component by sub-classing the `TDateTimePicker` component supplied with Delphi. The component displays a value of the type `TDateTime` and thus we need supporting classes that interface using this type. They are located in the unit `BoldDateTimeControlPack`. The complete `TBoldDateTimePicker`, with its supporting classes is shown in Diagram 1. As the figure shows the component is built by sub-classing `TDateTimePicker` and adding two helper-objects, a `TBoldElementHandleFollower` and a `TBoldDateTimeFollowerController`.

`TDateTimePicker` can operate in two modes, either as a Date picker, or as a Time picker. To keep down the complexity of the code most of the example assumes that the component is a Date picker. The additional code needed to support both modes is shown at the end.

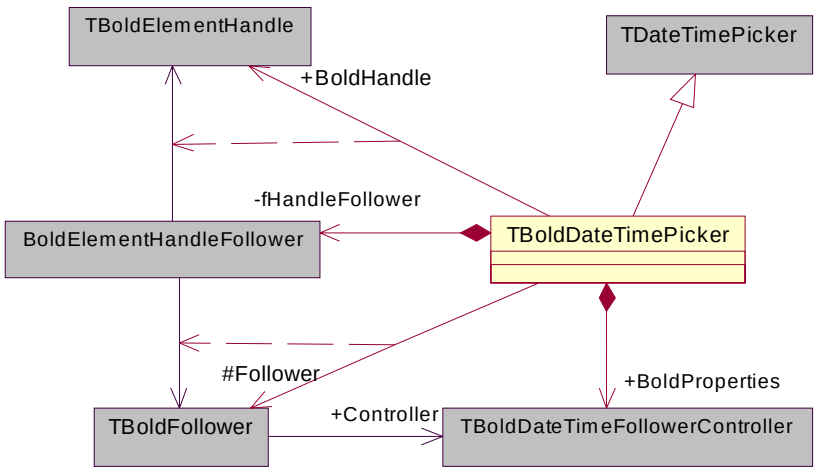

Diagram 1, Parts of a TBoldDateTimePicker

Creating the TBoldDateTimePicker class

The easiest way to create the actual component class is to use the component expert in Delphi. The result of running the expert is shown in Listing 1.

```
unit BoldDateTimePicker;

interface

uses
  Classes,
  ComCtrls;

type
  TBoldDateTimePicker = class(TDateTimePicker)
  end;

procedure Register;

implementation

procedure Register;
begin
  RegisterComponents('Bold Controls', [TBoldDateTimePicker]);
end;

end.
```

Listing 1 Result of running Expert

Adding the helper objects

The next step is to create the helper objects. This is done by overriding the constructor and destructor of the class as shown in **unit** BoldDateTimePicker;

interface

```
uses
  classes,
  SysUtils,
  comctrls,
  BoldUtils,
  BoldHandles,
  BoldControlPack,
  BoldElementHandleFollower,
  BoldDateTimeControlPack;

type
  TBoldDateTimePicker = class(TDateTimePicker)
  private
    fHandleFollower: TBoldElementHandleFollower;
    fBoldProperties: TBoldDateTimeFollowerController;
    function GetBoldHandle: TBoldElementHandle;
    procedure SetBoldHandle(const Value: TBoldElementHandle);
    procedure SetBoldProperties(const Value:
       TBoldDateTimeFollowerController);
    function GetFollower: TBoldFollower;
  protected
    property Follower: TBoldFollower read GetFollower;
  public
    constructor create(owner: TComponent); override;
    destructor destroy; override;
  published
    property BoldHandle: TBoldElementHandle read GetBoldHandle write
       SetBoldHandle;
    property BoldProperties: TBoldDateTimeFollowerController read
       fBoldProperties write SetBoldProperties;
  end;

procedure Register;

implementation


{ TBoldDateTimePicker }

constructor TBoldDateTimePicker.create(owner: TComponent);
begin
  inherited;
  fBoldProperties := TBoldDateTimeFollowerController.Create(self);
  fHandleFollower :=
       TBoldElementHandleFollower.Create(Owner,fBoldProperties);
end;

destructor TBoldDateTimePicker.destroy;
begin
  FreeAndNil(fHandleFollower);
  FreeAndNil(fBoldProperties);
  inherited;
end;

function TBoldDateTimePicker.GetBoldHandle: TBoldElementHandle;
begin
  result := fHandleFollower.BoldHandle;
end;

function TBoldDateTimePicker.GetFollower: TBoldFollower;
begin
  result := fHandleFollower.Follower
end;
```

```
procedure TBoldDateTimePicker.SetBoldHandle(const Value:
      TBoldElementHandle);
begin
  fHandleFollower.BoldHandle := Value;
end;

procedure TBoldDateTimePicker.SetBoldProperties(const Value:
      TBoldDateTimeFollowerController);
begin
  fBoldProperties.Assign(Value);
end;

procedure Register;
begin
  RegisterComponents( 'Bold Controls', [TBoldDateTimePicker] );
end;

end.
```

Listing 2. Note the parameters to the constructors. The one that may need some explanation is the first parameter to the HandleFollower. It should be the owner of the component, i.e. normally the form it is placed on.

If the component is installed and placed on a form, the object inspector will now show a `BoldProperties` property just as for a `TBoldEdit`. Note that the name `BoldProperties`, which is visible for the user of the component, reflects the role of the component from the perspective of the component user. The name of the class (`TBoldDateTimeFollowerController`) reflects the role of the component from the perspective of the component designer, i.e. you.

The set-method for the `BoldProperties` property calls `Assign`, and is needed in order to support forms inheritance in Delphi.

```
unit BoldDateTimePicker;

interface

uses
  classes,
  SysUtils,
  comctrls,
  BoldUtils,
  BoldHandles,
  BoldControlPack,
  BoldElementHandleFollower,
  BoldDateTimeControlPack;

type
  TBoldDateTimePicker = class(TDateTimePicker)
  private
    fHandleFollower: TBoldElementHandleFollower;
    fBoldProperties: TBoldDateTimeFollowerController;
    function GetBoldHandle: TBoldElementHandle;
    procedure SetBoldHandle(const Value: TBoldElementHandle);
    procedure SetBoldProperties(const Value:
       TBoldDateTimeFollowerController);
    function GetFollower: TBoldFollower;
  protected
    property Follower: TBoldFollower read GetFollower;
  public
```

```
    constructor create(owner: TComponent); override;
    destructor destroy; override;
  published
    property BoldHandle: TBoldElementHandle read GetBoldHandle write
       SetBoldHandle;
    property BoldProperties: TBoldDateTimeFollowerController read
       fBoldProperties write SetBoldProperties;
  end;

procedure Register;

implementation


{ TBoldDateTimePicker }

constructor TBoldDateTimePicker.create(owner: TComponent);
begin
  inherited;
  fBoldProperties := TBoldDateTimeFollowerController.Create(self);
  fHandleFollower :=
       TBoldElementHandleFollower.Create(Owner,fBoldProperties);
end;

destructor TBoldDateTimePicker.destroy;
begin
  FreeAndNil(fHandleFollower);
  FreeAndNil(fBoldProperties);
  inherited;
end;

function TBoldDateTimePicker.GetBoldHandle: TBoldElementHandle;
begin
  result := fHandleFollower.BoldHandle;
end;

function TBoldDateTimePicker.GetFollower: TBoldFollower;
begin
  result := fHandleFollower.Follower
end;

procedure TBoldDateTimePicker.SetBoldHandle(const Value:
       TBoldElementHandle);
begin
  fHandleFollower.BoldHandle := Value;
end;

procedure TBoldDateTimePicker.SetBoldProperties(const Value:
       TBoldDateTimeFollowerController);
begin
  fBoldProperties.Assign(Value);
end;

procedure Register;
begin
  RegisterComponents( 'Bold Controls', [TBoldDateTimePicker] );
end;

end.
```

Listing 2 Adding the helper objects

Making the component display values

The follower architecture will ensure that the value is kept current internally in the follower, and allow it to be accessed via the follower-controller. In order for the component to hook into this change, the follower-controller has two method pointers (a.k.a. events) that the component can plug into. `OnBeforeMakeUpToDate` will be called before the value stored in the follower is changed, and `OnAfterMakeUpToDate` will be called afterwards. Since we are only interested in displaying the value, we will hook the latter. Listing 3 shows how this is done by adding a private method _Display and assigning it to `BoldProperties.OnAfterMakeUpToDate`. The code in the method may look needlessly complex, and in this case, where there is only one follower, and one follower-controller, the code could simply have been written:

```
Date := BoldProperties.GetCurrentDateTime(Follower);
```

However, if you always use the form in Listing 3, you will avoid surprises when designing components that are more complex.

```
TBoldDateTimePicker = class(TDateTimePicker)
  private
  …
    procedure _Display(Follower: TBoldFollower);
  …
constructor TBoldDateTimePicker.Create(AOwner: TComponent);
begin
  …
  fBoldProperties := TBoldDateTimeFollowerController.Create(Self);
  fBoldProperties.AfterMakeUpToDate := _Display;
  …
end;

procedure TBoldDateTimePicker._Display(Follower: TBoldFollower);
var
  newDateTime: TDateTime;
begin
  newDateTime := TBoldDateTimeFollowerController(Follower.Controller).
       GetCurrentAsDateTime(Follower);
  if Date <> newDateTime then
    Date := newDateTime;
end;
```

Listing 3, _Display method

Making the component editable

In order to make the component two-way, the component must inform the follower of the fact that the user has supplied input that may have changed the value. This is done by calling `MayHaveChanged` on the follower-controller with the follower and the new value as parameters. (Note: it is still up to the follower-controller to decide if the value actually has changed, by calling `IsChanged` on the renderer for the component). Finding out when the component is modified can be quite tricky. Well-behaved components (such as `TEdit`) have a virtual method `Changed`, which is called when the value is changed. `TDateTimePicker` lacks a `Changed` method.

It does however have an event-handler `OnChange`. We cannot use the `OnChange` event directly since this would make it unavailable to the component user for its normal purpose. We can however use a trick: adding a new instance variable `fMyOnChange`, and publishing it under the name `OnChange`, effectively shadowing the `OnChange` property of the parent. When the user now sets the `OnChange` property he will actually be setting `fMyOnChange` rather that the parents `fOnChange`. We then add an event-handler `Change`, and hook it up in the `constructor`. Listing 4 shows the resulting code.

```
TBoldDateTimePicker = class(TDateTimePicker)
private
  …
  fMyOnChange: TNotifyEvent;
  procedure Change(sender:TObject);
  …
published
  …
  property OnChange: TNotifyEvent read fMyOnChange write fMyOnChange;
end;

constructor TBoldDateTimePicker.Create(AOwner: TComponent);
begin
  …
  inherited OnChange := Change;
end;

procedure TBoldDateTimePicker.Change;
begin
 if not (csDesigning in ComponentState) then
    BoldProperties.MayHaveChanged(date, Follower);
 if Assigned(fMyOnChange) then
   fMyOnChange(self);
end;
```

Listing 4, making the component editable

Implementing **bapExit**

In order for the `ApplyPolicy bapExit` to work the component must call `Apply` on the follower when it looses focus. This requires hooking the `CMExit` windows event. The code for this is shown in Listing 5.

```
uses
  controls,
  …

TBoldDateTimePicker = class(TDateTimePicker)
  private
    …
    procedure CMExit(var message: TCMExit); message CM_EXIT;
  …
  end;

procedure TBoldDateTimePicker.CMExit(var message: TCMExit);
begin
  if (BoldProperties.ApplyPolicy = bapExit) then
    Follower.Apply;
  DoExit;
```

```
end;
```
Listing 5, Implementing bamExit

Implementing ReadOnly on a component

Finally, we want to implement the correct behavior in deciding if a used shall be able to input values via a component or not. Many components have a `ReadOnly` property that can be used for this. Unfortunately, for some reason known only to the designers of VLC, `TDateTimePicker` lacks a `ReadOnly` property, so this section shows the code needed for another component (in this case `TBoldEdit`).

The behavior we want is to allow the setting of the "modifiability" of the component at design-time by setting the `ReadOnly` property, and then determining the effective "modifiability" in run-time.

This is done by a trick similar to the one used for `OnChange`. We add a field `fMyReadOnly`, and publish it as the property `ReadOnly`, effectively hiding the `ReadOnly` property of the parent. We then declare the inherited `ReadOnly` under the name `EffectiveReadOnly` (public), and make sure to set it at the same time as we redisplay the component. We further need to override the property representing the value of the component (`Text` in the case of a `TBoldEdit`), in order to prevent the value from being set programmatically. The code for all this is shown in Listing 6.

Since `TDateTimePicker` lacks a `ReadOnly` property the best we can do in this case is simply to add the property and ignore changes from the user by redisplaying the old value after each attempt. The code for this is shown Listing 7. In this case, we have guarded against setting the value programmatically, or against having set a value at design-time. This can be done in a similar way as in TBoldEdit, should it be needed.

```
TBoldCustomEdit = class(TCustomEdit)
  private
    …
    fMyReadOnly: Boolean;
    procedure SetReadOnly(value: Boolean);
    function GetEffectiveReadOnly: Boolean;
    …
  public
    property EffectiveReadOnly: Boolean read GetEffectiveReadOnly;
  published
    …
    property ReadOnly: Boolean read fMyReadOnly write SetReadOnly;
end;

constructor TBoldCustomEdit.Create(AOwner: TComponent);
begin
  …
  inherited ReadOnly := True;
end;
```

```
procedure TBoldCustomEdit.SetReadOnly(value: Boolean);
begin
  if fMyReadOnly <> value then
  begin
    fMyReadOnly := value;
    Follower.Display;  // Set inherited readonly correctly
  end;
end;

function TBoldCustomEdit.GetEffectiveReadOnly: Boolean;
begin
  Result := inherited ReadOnly;
end;

procedure TBoldCustomEdit.AfterMakeUptoDate(Follower: TBoldFollower);
begin
   …
  inherited ReadOnly := FMyReadOnly or not
       BoldProperties.MayModify(Follower);
  …
end;
```

Listing 6 , Adding ReadOnly capabilites to a component

```
TBoldDateTimePicker = class(TDateTimePicker)
  private
    …
    fReadOnly: Boolean;
    fEffectiveReadOnly: Boolean;
    procedure SetReadOnly(Value: Boolean);
  public
    …
    property EffectiveReadOnly: Boolean read fEffectiveReadOnly;
  published
    …
    property ReadOnly: Boolean read fReadOnly write SetReadOnly;
  end;

procedure TBoldDateTimePicker._Display(Follower: TBoldFollower);
begin
  …
  fEffectiveReadOnly := ReadOnly or
    not BoldProperties.MayModify(Follower);
end;

procedure TBoldDateTimePicker.Change;
begin
 if not (csDesigning in ComponentState) then
   if not EffectiveReadOnly then
     BoldProperties.MayHaveChanged(date, Follower)
   else
     Follower.Display; // Force redisplay of old value
 if Assigned(fMyOnChange) then
   fMyOnChange(self);
end;
```

```
procedure TBoldDateTimePicker.SetReadOnly(value: Boolean);
begin
  if fReadOnly <> value then
  begin
    fReadOnly := value;
    Follower.Display;  // Set EffectiveReadOnly correctly
  end;
end
```

Listing 7, Making TBoldDateTimePicker ReadOnly

Implementing Kind=dtkTime

In order to make the component work both as a date picker, and as a time picker we need to test the Kind property in two places and take different actions. This has been saved to last, in order not to mess up the code with low-lever details. Listing 8 shows the additions needed.

```
procedure TBoldDateTimePicker._Display(Follower: TBoldFollower);
var
  newDateTime: TDateTime;
begin
  …
  case Kind of
    dtkDate:
      if Date <> newDateTime then
        Date := newDateTime;
    dtkTime:
      if Time <> newDateTime then
        Time := newDateTime;
  end;  …
end;

procedure TBoldDateTimePicker.Change;
begin
   …
   if not EffectiveReadOnly then
     case Kind of
       dtkDate: BoldProperties.MayHaveChanged(Date, Follower);
       dtkTime: BoldProperties.MayHaveChanged(Time, Follower);
     end
   else
     Follower.Display; // Force redisplay of old value
   …
end;
```

Listing 8, implement Kind=dtkTime

Making the ocl-editor work

The ocl-editor relies on support from the component to figure out what the context of the ocl-expression is. Here is a listing of the code required to make it work:

```
type
  TBoldDateTimePicker = class(TdateTimePicker)
  private
    …
    procedure GetContextType: TBoldElementTypeInfo;
    …
  end;

procedure TBoldDateTimePicker.GetContextType: TBoldElementTypeInfo;
begin
  if assigned(BoldHandle) then
    result := BoldHandle.StaticBoldType
  else
    result := nil;
end;


constructor TboldDateTimePicker.Create(aOwner: Tcomponent);
begin
  …
  fBoldProperties := TBoldDateTimeFollowerController.Create(self);
  fBoldProperties.OnGetContextType := GetContextType;
  …
end;
```

Listing 9, Making the ocl-editor work