

15. Decorations

15.1. Prerequisites and Objectives

Before starting this chapter you should have:

1. A knowledge of programming in C++.
2. A knowledge of component interface design.

After completing this chapter you will have:

1. A knowledge of how to design Decoration components.

15.2. Introduction

Decorations are similar to UI Widgets, but normally do not provide user interaction. Despite the name, decorations often provide the user with useful information. Those that are purely decorative in nature tend to be the exception rather than the rule. For example a label that reads “Warning. If you proceed your hard-disk will be erased!” provides extremely useful information, despite the fact that the label component providing the information is a Decoration. Most decorations can be configured in some way.

We will provide two examples of Decorations, an LED component and a flashing label component.

15.3. The Methodology

Because Decoration components are seldom, if ever, purely decorative in nature, the first step in the design of a Decoration is to have a clear idea of its purpose. For example, the purpose of a label is to provide static textual information, while the purpose of a progress bar is to indicate what percentage of a task has been completed. Once a purpose

has been clearly specified, the next step is to design a visual display to meet that purpose. It is generally useful to complete a set of drawings during this step.

Decorations are not always static. If some form of data input is required (such as text for a label) it should be specified through properties. State changes should also be specified using properties. The use of properties allows these values to be specified at design time. Generally speaking, once the visible elements of the component have been designed, the rest of the design is quite simple.

15.4. A Simple LED Component

The LED component displays a picture of an LED. The bitmaps for this component were created on a white background. Because the bitmap editor antialiased the outline of the bitmaps with the white background, the LED's look strange on any background that is not white. On the other hand, they look *excellent* on white backgrounds.

The LED component has two properties, one to set the color of the LED and one to set the color of the background. The LED bitmaps are circular, but are drawn in a square background just large enough to contain the component. Figure 15.1 shows the three different bitmaps used to display the LED, while Figure 15.2 gives the description of the LED properties.



Figure 15.1. The LED Bitmaps.

Property Design Table		
Name	Type	Function
BackColor	Color Default=Gray	Determines the background color of the rectangle containing the LED bitmap.
LedColor	Enumerated Led_Red=0 Led_Green=1 Led_Yellow=2 Default=Led_Red	Determines the color of the LED. If a value other than 0, 1, or 2 is assigned, the value Led_Red will be assigned.

Figure 15.2. The LED Property Design Table.

The drawing routine performs the following operations. It determines the current value of the *LedColor* property, and loads the proper bitmap for the color. (The bitmaps are stored as resources in the component module.) The bitmap is copied into the component window, and the white border is flood-filled with the background color. Because the implementation of the component is straightforward, we omit the details. The code for the LED component is available on the CD ROM.

15.5. The Flashing Text Control

The flashing text control is a text label that has the capability of flashing to call attention to the text message. The programmer has the capability of changing the background color, and the on and off text colors. The default state of the text message is on. When the text is flashing, the message alternates between the on and off state. The programmer can set separate text colors for each state or specifying that the text is invisible in either the on or the off state. The programmer can also set the state of the text to Flashing or Not-Flashing.

When the Flashing state is first set, the component starts a system timer with an interval of 250 ms (1/4 second). This timer fires an event each time the 250 ms interval

expires. This event is used to toggle the on/off state of the text. When the flashing state is set to Not-Flashing, the text returns to the on state.

The **CTextControl** class given in Figure 15.3 was created to simplify the development of the Flashing Text component. This class encapsulates all information about the flashing text. The variables of this class are modified directly by the component routines.

```
class CTextControl
{
public:
    COLORREF BackColor;
    UINT TimerID;
    BOOL OffInvis;
    BOOL OnInvis;
    BOOL Flashing;
    BOOL NowOn;
    COLORREF OffColor;
    COLORREF OnColor;
    CString Value;
    CTextControl();
    virtual ~CTextControl();
};
```

Figure 15.3. The CTextControl Class.

With two exceptions, the elements of the **CTextControl** object are available as properties of the Flashing Text component. The variable *TimerID* is used to communicate with the system timer that controls the flashing, while the variable *NowOn* is set during the flashing process to control the display of the text. The heart of the Flashing Text component is the drawing routine which is given in Figure 15.4.

```

void CFlashTextCtrl::OnDraw(
    CDC* pdc, const CRect& rcBounds, const CRect& rcInvalid)
{
    // paint the background
    CBrush MyBack;
    MyBack.CreateSolidBrush(Txt.BackColor);
    CBrush * OldBrush = pdc->SelectObject(&MyBack);
    pdc->FillRect(rcBounds, &MyBack);
    // save text-related colors
    COLORREF OldColor = pdc->GetTextColor();
    COLORREF OldBackClr = pdc->GetBkColor();
    // set text background color
    pdc->SetBkColor(Txt.BackColor);
    // set text foreground color and draw
    if (Txt.NowOn)
    {
        if (!Txt.OnInvis)
        {
            pdc->SetTextColor(Txt.OnColor);
            pdc->TextOut(0,0,Txt.Value);
        }
    }
    else
    {
        if (!Txt.OffInvis)
        {
            pdc->SetTextColor(Txt.OffColor);
            pdc->TextOut(0,0,Txt.Value);
        }
    }
    // restore drawing state.
    pdc->SelectObject(OldBrush);
    MyBack.DeleteObject();
    pdc->SetTextColor(OldColor);
    pdc->SetBkColor(OldBackClr);
}

```

Figure 15.4. The Flashing Text OnDraw Routine.

Each time the component is redrawn, the existing text is erased and redrawn. The current on/off state of the text is used to determine the color used to draw the text. The flashing state of the component is controlled by the *Flash* property, the implementation of which is given in Figure 15.5.

```

void CFlashTextCtrl::SetFlash(BOOL bNewValue)
{
    if (bNewValue)
    {
        if (!Txt.Flashing)
        {
            Txt.Flashing = TRUE;
            Txt.TimerID = CWnd::SetTimer(1,250,NULL);
            Txt.NowOn = TRUE;
            CWnd::Invalidate();
        }
    }
    else
    {
        if (Txt.Flashing)
        {
            Txt.Flashing = FALSE;
            Txt.NowOn = TRUE;
            CWnd::KillTimer(1);
            CWnd::Invalidate();
        }
    }
    SetModifiedFlag();
}

```

Figure 15.5. Implementation of the Flash Property.

The *OnTimer* routine of the support class is used to process events from the timer, and to control the flashing of the text. While the text is flashing, these events will occur at a rate of approximately four times per second. The implementation of the *OnTimer* routine is given in Figure 15.6.

```

void CFlashTextCtrl::OnTimer(UINT nIDEvent)
{
    if (Txt.NowOn)
    {
        Txt.NowOn = FALSE;
    }
    else
    {
        Txt.NowOn = TRUE;
    }
    CWnd::Invalidate();
    COleControl::OnTimer(nIDEvent);
}

```

Figure 15.6. The Flashing Text OnTimer Routine.

Except for the *Flash* property, the implementations of the other properties of the Flashing Text component are straightforward. Figure 15.7 gives the description of the properties of the Flashing Text component.

Property Design Table		
Name	Type	Function
BackGround	Color Default=Gray	Determines the background color of the rectangle containing the text.
OnColor	Color Default = White	The color of the text when it is not flashing.
OffColor	Color Default = Black	When flashing, text color alternates between the OnColor and the OffColor.
Value	String Default="Flashing Text"	The text displayed by the component.
InvisibleWhenOn	Boolean Default = FALSE	If true, the text is invisible when the component is not flashing, and when the state of the text is <i>on</i> .
InvisibleWhenOff	Boolean Default = FALSE	If true, the text is invisible when <i>off</i> . The text will appear to flash on and off.
Flash	Boolean Default=FALSE	If true the text is flashing, if false, the text is in the <i>on</i> state.

Figure 15.7. The LED Property Design Table.

All of the properties listed in Figure 15.7 are persistent properties. This implies that the text may be in the flashing state when the component is initialized, which in turn implies that the timer that controls the flashing must be started during the initialization process rather than as a consequence of assigning a value to the *Flash* property. The *OnCreate* routine of the support class is called after the values of persistent properties have been restored, but before the component is displayed. This is the ideal place to start the timer. The *OnCreate* routine is illustrated in Figure 15.8.

```
int CFlashTextCtrl::OnCreate(LPCREATESTRUCT lpCreateStruct)
{
    if (COleControl::OnCreate(lpCreateStruct) == -1)
    {
        return -1;
    }
    if (Txt.Flashing)
    {
        Txt.TimerID = CWnd::SetTimer(1,250,NULL);
        Txt.NowOn = TRUE;
        CWnd::Invalidate();
    }
    return 0;
}
```

Figure 15.8. The OnCreate Routine of the Flashing Text Component.

15.6. A Review of the Methodology

The main design element for a Decoration is the design of the visual display. The main question you want to answer is, “What will it look like?” In your formal design documents you should have a clear English description of the visual aspects of the component, accompanied by hand drawings, if necessary. (Hand drawings would be good for the LED component, but pointless for the Flashing Text component.)

Next, it is necessary to determine whether the display will be static or configurable. If the display is static, virtually no component interface will be required. If it is configurable, then a set of configuration properties and methods must be designed. Persistent properties are preferred for this purpose, because this allows the component to be configured at design time.

For many decorations, the design of the visible elements will be the most time-consuming portion of the design (this was certainly the case for the LED component.) Once the visible elements have been designed, the rest of the component design should be straightforward.

15.7. Conclusion

The primary purpose for a Decoration is to supply information to the end-user of a program. There are Decoration components that serve a purely decorative function, but these tend to be the exception rather than the rule. If you have some new, interesting way of passing information to the user that does not require any user interaction, then you should create a Decoration component to present the information. Decorations have the most in common with Display components.

15.8. Exercises

1. Create a speedometer component that will display a picture of a speedometer with markings from 0 to 100. A configuration parameter will be used to set a value from 0 to 100 indicating the position of the needle. The OnDraw routine will draw the picture with the needle in the correct position.
2. Create a progress bar component similar to that which you see when you install software or download something.
3. Create an invisible decoration that can be used to produce several different “beep” tones.