

# 14. UI Widgets

## ***14.1. 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 user interface components.
2. A knowledge of the UI Widget design methodology.

## ***14.2. Introduction***

At one time, virtually all components were User Interface Widgets. Even today, the UI Widget is the predominant type of commercial component. Some UI Widgets, like the list box, have programmable behavior, while others, like the button, do not. Because so many UI Widgets are available, it is unlikely that you will need to implement any of your own, however, it is still good to understand the principles of such components. Many interesting programs can be created using nothing but UI Widgets.

The primary function of a UI Widget is to transform mouse-clicks into programmable events. (Do not be confused by text boxes. Text boxes are editors, not UI Widgets.) In addition to firing events, UI Widgets can display useful information, and provide visual feedback regarding the events they are creating. Consider, for example, the scroll-bar component. The slider portion of the scroll bar provides useful information about the

current position in a document, and also provides visual feedback about scrolling operations.

In this chapter we will give examples of both configurable and non-configurable components.

### **14.3. The Methodology**

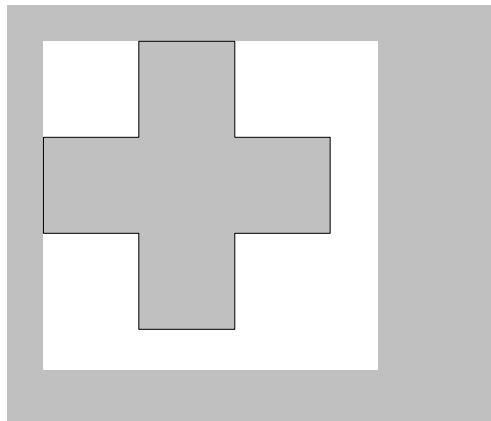
The first step in designing a User Interface Widget is the design of the visible display. You must decide between using a pre-drawn bitmap or drawing the display using drawing functions. If useful information is to be displayed along with static pictures, one must determine the source of that information. The typical way to handle this is by supplying the data through a property, but complex data can also be supplied through a series of method calls.

The second step is determining how the user will interact with the display. You must specify the hot spots of the display. These are the spots that will produce events when clicked by the user. If keyboard shortcuts are to be used, you must determine the meaning of the individual keystrokes, and how these keystrokes will equate to mouse clicks on the display. For example, if your Widget displays a left-pointing arrow, pressing the left arrow button might be the equivalent of clicking the left-pointing arrow.

The final step is to design the event structure of the Widget. The primary purpose of a UI Widget is to transform and transmit user input to the host program. This must be done through events. At the very least, it is generally necessary to analyze the mouse coordinates to determine which portion of the display has been clicked. The result of this design step will be a list of events, a set of parameters for each event along with the meaning of each parameter, and a list of conditions that trigger each event.

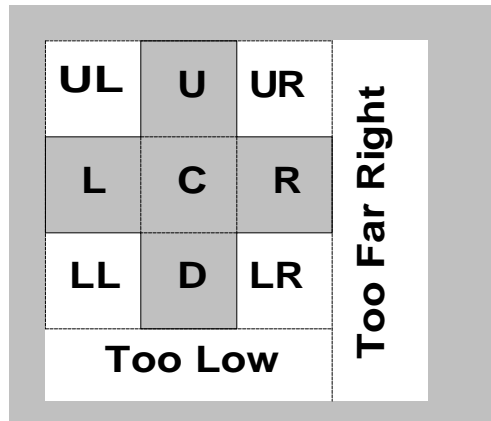
## 14.4. The Cross Control

Our first example is a model of the four-way button that is available on some video-game controllers. The four-way button is shaped like a cross, and permits movement in four directions, up, down, left and right. For some controllers, diagonal movement is possible by pushing two buttons at the same time. We will use a simple drawing to represent the control, as shown in Figure 14.1. Figure 14.1 shows the component as it would appear in a Visual Basic program. Normally the Visual Basic programmer will adjust the white area to be the same size as the cross, however this is not necessary. The component window includes the entire white area, and the cross within it.



**Figure 14.1. The Cross Component.**

The component window is divided into eleven “hot zones,” each one of which is associated with a particular event. These zones are shown in Figure 14.2. Clicking in one of the hot zones will produce the associated event.



**Figure 14.2. The Cross Component Hot Zones.**

Unlike most of the other types of components, for which events are almost incidental, the event description is the most important part of the definition of a UI Widget. Because the cross component has so many different events, we will use an event table to describe them, instead of an individual description for each event. Figure 14.3 gives the event table for the cross component.

Event Table	
<b>Description</b>	These events report hot-zone clicks.
<b>Arguments</b>	Void
<b>Name</b>	<b>Triggers</b>
Center	Mouse Click in the center of the cross.
Down	Mouse Click on the lower arm of the cross.
Up	Mouse Click on the upper arm of the cross.
Left	Mouse Click on the left arm of the cross.
Right	Mouse Click on the right arm of the cross.
DownLeft	Mouse Click between the lower arm and the left arm.
DownRight	Mouse Click between the lower arm and the right arm.
UpLeft	Mouse Click between the upper arm and the left arm.
UpRight	Mouse Click between the upper arm and the right arm.
TooFarRight	Mouse Click to the right of the extension of the right arm.
TooLow	Mouse Click below the extension of the lower arm.

**Figure 14.3. Event Table for the Cross Component.**

Because the UI Widget provides a visual interface with the user of the program, considerable effort should be made to create a pleasing display. (Admittedly, we have failed somewhat in this regard.) In many ways the drawing of the component is more important than the event structure. If specific meanings are to be associated with certain user actions, it must be clear to the user what to do. One way we might improve the cross component is to draw an arrowhead on each arm of the cross. However, we will leave this as an exercise for the reader. The drawing routine that produces the cross is given in Figure 14.4. This routine draws the outline of the cross, and flood-fills the interior with gray.

```

void CCrossCtrl::OnDraw(
    CDC* pdc, const CRect& rcBounds, const CRect& rcInvalid)
{
    // set background to white
    pdc->FillRect(rcBounds, CBrush::FromHandle(
        (HBRUSH)GetStockObject(WHITE_BRUSH)));
    // set up colors
    CBrush * GreyBrush = CBrush::FromHandle(
        (HBRUSH)GetStockObject(LTGRAY_BRUSH));
    CBrush * OldBrush = pdc->SelectObject(GreyBrush);
    CPen * BlackPen = CPen::FromHandle((HPEN)GetStockObject(BLACK_PEN));
    CPen * OldPen = pdc->SelectObject(BlackPen);
    // draw outline
    pdc->MoveTo(50,0);
    pdc->LineTo(100,0);
    pdc->LineTo(100,50);
    pdc->LineTo(150,50);
    pdc->LineTo(150,100);
    pdc->LineTo(100,100);
    pdc->LineTo(100,150);
    pdc->LineTo(50,150);
    pdc->LineTo(50,100);
    pdc->LineTo(0,100);
    pdc->LineTo(0,50);
    pdc->LineTo(50,50);
    pdc->LineTo(50,0);
    // fill center -- RGB(0,0,0) is color to be replaced
    pdc->FloodFill(75,75,RGB(0,0,0));
    // clean up color selections
    pdc->SelectObject(OldBrush);
    pdc->SelectObject(OldPen);
}

```

**Figure 14.4. The OnDraw Function of the Cross Component.**

We will make the cross buttons work like a standard button. In a standard button, the click event is not issued until the mouse button is released. If the user clicks on the button, and then holds the mouse-button down and moves the mouse cursor off the button, no click event is issued. When the button goes down, we will record that the button has been pressed, and then wait for it to be released. Figure 14.5 shows the actions taken when the button is pressed.

```
void CCrossCtrl::OnLButtonDown(UINT nFlags, CPoint point)
{
    // record the fact that the mouse button has been pressed
    ButtonIsDown = 1;
    // we want to see the mouse-up, even if it happens outside
    // the control window.
    CWnd::SetCapture();
    // continue with normal event processing
    COleControl::OnLButtonDown(nFlags, point);
}
```

**Figure 14.5. Button Down Processing for the Cross Component.**

The real event processing takes place when the button is released. Figure 14.6 shows the processing for the mouse-up event. Note that because of the mouse-capture, this event will be processed, even if the mouse-button is released outside the component window. The mouse-up routine first tests to see if the mouse was clicked inside the component window, in which case *ButtonIsDown* will be set to one. If not, no processing takes place. Next, it checks to see if the current mouse cursor position is inside the component window. If not, no processing takes place. Finally the mouse position is analyzed. The first check is for the Too Low and Too Far Right zones. If the mouse is not in either of these zones, it must be within the drawing of the cross. The first check determines the horizontal position of the mouse. Once that has been determined, an additional check is done to determine the vertical position of the mouse within the column. These two pieces of information will determine the zone in which the mouse appears. The appropriate event for the zone is then fired.

```

void CCrossCtrl::OnLButtonUp(UINT nFlags, CPoint point)
{
    // do nothing if Mouse-Down did not occur inside component window
    if (ButtonIsDown)
    {
        // release mouse capture
        // strange stuff will happen if we forget this
        ReleaseCapture();
        RECT R;
        CWnd::GetClientRect(&R);
        // do nothing if mouse-up is outside the component window
        if (point.x >= R.left && point.x <= R.right &&
            point.y >= R.top && point.y <= R.bottom)
        {
            // first check for the illegal positions
            if (point.x > 150)
            {
                FireTooFarRight();
            }
            else if (point.y > 150)
            {
                FireTooLow();
            }
            // Now find out which column the mouse is in
            else if (point.x >= 0 && point.x < 50)
            {
                // Left column, now check for the row
                if (point.y >= 0 && point.y < 50)
                {
                    FireUpLeft();
                }
                else if (point.y >= 50 && point.y <= 100)
                {
                    FireLeft();
                }
                else
                {
                    FireDownLeft();
                }
            }
            else if (point.x >= 50 && point.x <= 100)
            {
                // Middle column, now check for the row
                if (point.y >= 0 && point.y < 50)
                {
                    FireUp();
                }
                else if (point.y >= 50 && point.y <= 100)
                {
                    FireCenter();
                }
                else
                {
                    FireDown();
                }
            }
            else

```

```
    {
        // Right column, now check for the row
        if (point.y >= 0 && point.y < 50)
        {
            FireUpRight();
        }
        else if (point.y >= 50 && point.y <= 100)
        {
            FireRight();
        }
        else
        {
            FireDownRight();
        }
    }
}
// weirdness will happen if we forget this
ButtonIsDown = 0;
// continue with normal event processing
COleControl::OnLButtonUp(nFlags, point);
}
```

**Figure 14.6. The Button-Up Routine for the Cross Component.**

An additional feature we could add to this component is visual feedback about which portion of the cross has been clicked. This will require tracking the Mouse-Move events, because the event that is eventually fired will be that corresponding to the final position of the mouse rather than its initial position. The visual display of the cross component is its greatest weakness. A considerable amount of work will be required to bring the display up to commercial quality.

### **14.5. The Cards Component**

Unlike the cross component, the cards component will provide a useful visual display. This is accomplished by borrowing some drawings from an existing operating system component. The Windows operating system includes a file named “cards.dll” which contains high-quality drawings of a deck of playing cards, along with several designs for the backs of the cards, and a couple of other useful drawings. This file has been used by

many programmers to create interactive card games. Rather than provide a simple wrapper for the cards.dll drawings, we will add features that enhance the utility of the component, particularly for creating stacks of cards.

To begin with, we will specify two modes of operation for our component, single-card mode and multiple-card mode. In single-card mode, we want the user to be able to specify the card to be displayed. We also want the component to size itself to a single card, so that there will be no extra white space as there was with the cross component. Figure 14.7 shows what the display should look like for four single-card components.



**Figure 14.7. Single-Card Mode.**

Since many card games require one to select a card before moving it, we also want some visible indication that a card is selected. To do this, we will add a *Selected* property to the component, and invert the colors of the card when it is displayed. (This is a fairly simple graphical operation.)

In multiple-card mode, we want the cards to be displayed as a stack, as in Figure 14.8. We will permit the user to determine the amount of space between each card, which will determine the visible portion of the lower cards. When a stack is selected, we will invert the display of the top card only. (Modifying the component to permit the selection of a particular card in the stack is an exercise left for the reader.)



**Figure 14.8. A Card Stack.**

In multiple-card mode we want to be able to add and delete cards from the stack, and when we do this, we want the component to resize itself to fit the new stack of cards. Of course, we want to be able to access the members of the stack to determine which card is in a particular position, and we especially want to be able to access the top card in the stack. We also want to be able to determine how many cards are in the stack. To accomplish our goals, we have created a set of properties that will allow us to control the display of the component. Figure 14.10 gives the description of these properties.

Property Design Table		
Name	Type	Function
<b>Multiple</b>	Boolean Default=FALSE	Determines whether the component is in single-card mode (FALSE) or multiple card mode (TRUE).
<b>Selected</b>	Boolean Default=FALSE	Determines whether the card, or the stack is selected. This property will invert the colors of the card, or the top card in the stack.
<b>CardValue</b>	Long Integer Default=1 Minimum=0 Maximum=100	Determines which card face will be displayed. Each suit is numbered in Ace, 2-10, Jack, Queen, King order. The suits are in bridge order: Clubs, Diamonds, Hearts, Spades. 1 is the ace of clubs, 52 is the King of Spades. Numbers higher than 52 display card backs and place holders.
<b>CardCount</b>	Long Integer Default=0 Minimum=0 Maximum=None	Determines the number of cards in the stack when in multiple-card mode. This property is active regardless of the current display mode
<b>Card</b>	Long Integer Array Default Size = 0 Element Default=0 Element Minimum=0 Element Maximum=100	Determines the value of each card in the stack. The size of the array is equal to the CardCount property. The array is indexed from 0, with Index 0 being the lowest card in the stack. The top card in the stack is Card[CardCount-1].
<b>Offset</b>	Long Integer Default=30 Minimum=0 Maximum=100	Determines the number of pixels between the tops of each successive card in the stack. The offset of 30 is enough for the number and suit of the lower cards to be visible. Some games pack the cards more tightly.

**Figure 14.9. The Cards Component Property Design Table.**

Because we are creating a UI Widget, we also need to be concerned about the event structure. We will provide a single event, *Click*, that will notify the container that the component has been clicked. It would make sense to add a parameter to this event to determine which card in a stack has been clicked, but we will leave this as an exercise for the reader. The event definition for the *Click* event is given in Figure 14.10.

Event Description			
<b>Name</b>	Click		
<b>Description</b>	Notifies the container that the component has been clicked.		
<b>Triggers</b>	Mouse-Click Anywhere in the component window.		
Arguments	Name	Type	Description
Void			

**Figure 14.10. Click Event Description for the Cards Component.**

In addition to providing default values for properties and other support class variables, the initialization step of the Cards component must also load the cards.dll file. This is done in three steps. First, the initialization routine attempts to load cards.dll. If that fails, it then tries to load cards32.dll. If that fails, it uses the default images in the Cards component. Initialization also sets the size of the component, and flags the component as non-resizable. If the initialization step loads cards.dll or cards32.dll, the termination step will unload the file.

The drawing routine is somewhat complicated, but straightforward. The first step is to determine the current drawing mode. In single-card mode the drawing routine loads a single bitmap and copies it into the component window. In multiple-card mode, this step is repeated for each card. As each card is drawn, the upper and lower edges of the target rectangle are incremented by the value of the *Offset* property. The standard Windows operating system function BitBlt is used to copy the bitmap images into the component window. This function has several drawing modes, two of which are Source-Copy, and Invert-Source-And-Copy. For non-selected components, only the Source-Copy mode is used. For selected components (Selected property equal to TRUE), the Invert-Source-And-Copy mode is used for the last card drawn.

Although the property implementations are straightforward, the *write* portion of many of them is complicated by the necessity of resizing the component. The resize operation

is required when changing the value of the *Multiple*, *CardCount*, and *Offset* properties. (See the accompanying CD for details on how this is done.)

The *write* portion of the *CardCount* property is especially complicated, because the internal array of card values must be destroyed and recreated. To avoid placing too many demands on the user of the component, the data in the existing card value array must be copied into the new card value array. If the new array is smaller than the old, the extra values are discarded. If the new array is larger than the old, the new positions are filled with zeros.

At this point the Cards component has everything it needs to be used as a component in a card-game program. However, there is one additional feature that we would like to add to enhance its utility in such programs. In the real world virtually every game of cards begins with a shuffle and a deal. This is also true for simulated card games, so anyone who constructs a card-game program must also provide some method for shuffling and dealing cards. We can greatly enhance the utility of the Cards component by building such a feature into it. To this end, we add two methods to the Cards component, *Deal*, and *DrawCard*. The *Deal* method shuffles a deck of 52 cards, and deals them into an array. The formal description of these methods is given in Figure 14.11. The *DrawCard* method draws cards from the array one at a time. These functions can be used to deal bridge hands using the Visual Basic routine shown in Figure 14.12. In this routine, *Deck* is an instance of the *Cards* control, while *North*, *East*, *West*, and *South* are instances of the Queue component described in Chapter 12.

Method Description			
<b>Name</b>	Deal		
<b>Return Value</b>	Void		
<b>Description</b>	Creates a 52-element array containing the numbers from 1-52, and sorts the array into random order.		
<b>Arguments</b>	<b>Name</b>	<b>Type</b>	<b>Description</b>
Void			

Method Description			
<b>Name</b>	DrawCard		
<b>Return Value</b>	Long Integer		
<b>Description</b>	Draws a card from the randomized array of card numbers created by the <i>Deal</i> method. The <i>Deal</i> method must be called before calling <i>DrawCard</i> .		
<b>Arguments</b>	<b>Name</b>	<b>Type</b>	<b>Description</b>
	SlotNumber	Long Integer	Element of the randomized array of card numbers. SlotNumber must be taken from the range 0-51. If <i>Deal</i> has not been called or if the SlotNumber is not in the range 0-51, zero will be returned.

**Figure 14.11. Method Descriptions for Deal and DrawCard.**

```

Deck.Deal
For i = 0 To 12
    North.Value = Deck.DrawCard(i)
Next i
For i = 13 To 25
    East.Value = Deck.DrawCard(i)
Next i
For i = 26 To 38
    West.Value = Deck.DrawCard(i)
Next i
For i = 39 To 51
    South.Value = Deck.DrawCard(i)
Next i
    
```

**Figure 14.12. Dealing Bridge Hands.**

The implementation of the *Deal* method appears in Figure 14.13. This method first checks for the existence of the *CardDeck* array. (*CardDeck* is a support-class variable.) If it does not exist, it is created. The array is initialized with the numbers 1-52 in ascending

order. A second 52-element array, *Work*, is initialized by assigning a random number to each element. (Since *Work* is a local variable it will be destroyed when the *Deal* method terminates.) The two arrays are then sorted in parallel, using the *Work* array as the key. The Insert Sort algorithm is used to perform the sort. The random number seed is initialized from the system clock when the component is initialized, so each call to *Deal* will produce a different randomization of the deck.

```
void CCardsCtrl::Deal()
{
    long Work[52]; // key for the random sort
    if (CardDeck == NULL)
    {
        CardDeck = new long [52]; // create if it doesn't exist
    }
    for (long i=0 ; i<52 ; i++)
    {
        Work[i] = rand(); // random key
        CardDeck[i] = i+1; // card number
    }
    for (i=1 ; i<52 ; i++) // sort by the random key
    {
        long x = Work[i];
        long y = CardDeck[i];
        for (long j=i-1 ; j>=0 ; j--)
        {
            if (Work[j] < x)
            {
                break;
            }
            Work[j+1] = Work[j];
            CardDeck[j+1] = CardDeck[j];
        }
        j++;
        Work[j] = x;
        CardDeck[j] = y;
    }
}
```

**Figure 14.13. The Deal Routine.**

## 14.6. The Dice Component

Our final example is a simulation of a six-sided die. The actions of the die are modeled by the class **CDie**. The Dice component serves as a wrapper for the **CDie** class. It is not clear whether this component should be classified as a UI Widget or as a Model. This serves to illustrate that the dividing line between categories is not always clear.

Figure 14.14 gives the definition of the **CDie** class. Each face of the die is represented by a short integer. The value of each face must be taken from the range 1-6, and it is necessary that the faces have the correct relationship to one another. A quick examination of a die will reveal the following properties. The sum of the numbers on two opposing faces is always equal to seven. Thus 1 must be opposite 6, 2 opposite 5, and 3 opposite 4. There are two different configurations of spots that will meet this requirement. The correct configuration can be determined by looking at the corner where 1, 2, and 3 meet. When looking down at this corner, the numbers must appear in counter-clockwise order. Thus, if the die is in front of you with 1 at the top, and 2 to the left, 3 must be facing you. All operations on the die, including initialization, must maintain these relationships. To enforce these rules, the six faces are maintained in protected variables. The *Get* functions can be used to obtain the value of each face. The *Reset* function, which is called by the class constructor, puts 1 at the top and 3 to the front with all other faces initialized consistently. The functions *RollRight*, *RollLeft*, *RollFront*, *RollBack*, *SpinLeft*, and *SpinRight* turn the die 90 degrees in the indicated direction. For *RollRight* and *RollLeft*, the front and back are left unchanged, for *RollFront* and *RollBack*, the left and right are left unchanged, while for *SpinLeft* and *SpinRight*, the top and bottom are left unchanged. The *SetTop* and *SetFront* functions can be used to put the die in any desired position.

Both functions must be called, with *SetTop* being called first. *SetTop* moves the specified number to the top, while *SetFront* spins the die without changing the top or bottom. *SetFront* will fail if the specified face is on the top or the bottom. We will omit the implementation details of this class because they are straightforward and somewhat tedious. The full implementation appears on the accompanying CD ROM.

```
class CDie
{
protected:
    short Back;
    short Front;
    short Right;
    short Left;
    short Bottom;
    short Top;

public:
    CDie();
    virtual ~CDie();

    short GetBack(void);
    short GetFront(void);
    short GetRight(void);
    short GetLeft(void);
    short GetBottom(void);
    short GetTop(void);

    void SetFront(short Face);
    void SetTop(short Face);

    void Reset(void);

    void SpinRight(void);
    void SpinLeft(void);
    void RollBack(void);
    void RollFront(void);
    void RollRight(void);
    void RollLeft(void);
};
```

**Figure 14.14. The CDie Class.**

The Dice control provides properties to access each face (*TopFace*, etc.) as well as wrapper methods for each of the class manipulator functions. In addition to these, it

provides two properties, *DotColor* and *FaceColor*, to set the dot color and the face color of the die, and a method, *Toss*, that can be used to simulate a random throw of the die.

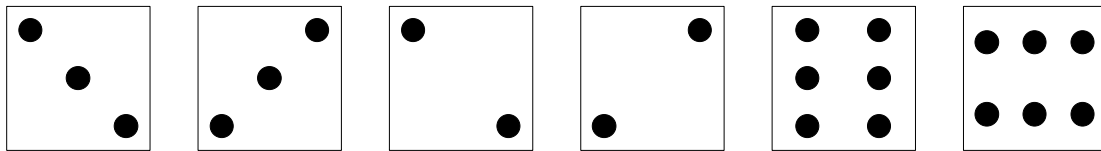
The only interesting implementation is that of the *Toss* method illustrated in Figure 14.15. The *Toss* method uses the *SetTop*, *SpinLeft*, and *SpinRight* functions of the **CDie** class to randomly position the die. The top is chosen randomly by generating a random number in the range 1-6. The front face is positioned randomly by generating a random number in the range 0-3. This number is used to select one of the four faces, Front, Back, Left and Right as the new front face. The spin functions are used to bring the selected face to the front.

```
void CDiceCtrl::Toss()
{
    Die.SetTop((short)((rand()%6)+1));
    long Move = rand()%4;
    switch (Move)
    {
        case 0:
        {
            Die.SpinLeft();
        }
        break;
        case 1:
        {
            Die.SpinRight();
        }
        break;
        case 2:
        {
            Die.SpinLeft();
            Die.SpinLeft();
        }
        break;
    }
    CWnd::Invalidate();
}
```

**Figure 14.15. The Toss Method.**

There is one additional problem that we face if we are to provide an accurate simulation of a real die, namely the configuration of the dots on faces 2, 3, and 6. These

faces can be drawn in two different ways. (Faces 1, 4, and 5 are symmetric and do not have this problem.) Figure 14.16 illustrates the different drawings for the 2, 3, and 6 faces.



**Figure 14.16. Die Face Orientations.**

To determine which drawing to use in each case, it is necessary to examine a real die. Such an examination will reveal the following. When 1 is at the top, 3 will lean to the left, and 2 will lean to the right. These are the second and third drawings in Figure 14.16. Furthermore, the open ends of the 6 point at 3 and 4. The drawing routine will draw only the top face of the die. To determine which orientation of 2, 3, and 6 must be drawn, it is necessary to examine the front face. Apart from these details, the drawing of the die is simply a matter of drawing a few circles.

The die component has one event, *Click*, that is fired when the user clicks anywhere in the component window. The description of this event is given in Figure 14.17.

Event Description			
<b>Name</b>	Click		
<b>Description</b>	Notifies the container that the component has been clicked.		
<b>Triggers</b>	Mouse-Click Anywhere in the component window.		
<b>Arguments</b>	<b>Name</b>	<b>Type</b>	<b>Description</b>
Void			

**Figure 14.17. Click Event Description for the Dice Component.**

## **14.7. A Review of the Methodology**

Most UI Widgets combine information display with user interaction. (Buttons are the exception, but we already have a lot of those.) In the design of a UI Widget, you must answer three questions: what information will be displayed, how will it be displayed, and how will the user interact with the display? The first step is to design the visible elements of the component, and the properties and methods used to control the visible interface. Next it is necessary to design the event structure of the component. When designing an event, it is necessary to have a clear picture of how the programmer will use the event. It is necessary to know what information the programmer will need to process the event, and what actions the programmer would normally be expected to make in response to the event. Once these elements have been designed, hooking the events to the visible elements of the control is a straightforward process. When a mouse button is clicked within the component window, the component must analyze the mouse-cursor position to determine the portion of the visible display that has been clicked. This analysis, along with an analysis of the current state of the component will permit the component to select and fire the proper event.

## **14.8. Conclusion**

The UI Widget is the most common type of component available today. There are so many UI Widgets available, that it is difficult to find something that hasn't already been done. The wide availability of existing components makes it unlikely that the component designer will need to many create new ones.

## **14.9. Exercises**

1. Create a UI Widget for an automatic transmission indicator. The user should be able to “shift gears” by clicking on one of the characters P, R, N, D, 1, or 2.
2. Create a three-way selector control that permits the user to select one of three states by clicking on one arm of a T. The T should look something like the following drawing. Draw a black square on the arm that is currently selected.
3. Create a component similar to the cross control, but use an 8-pointed star to indicate directions. Clicking on the appropriate point makes a move in the indicated direction.
4. Modify the cross control so the display is three dimensional, an moves down when pressed.
5. Modify the cross control so that holding the mouse button down in a particular location will create a series of events instead of a single event. The events should occur at a rate of 10/sec.