

6. Models

6.1. 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 wrap an object model in a component wrapper.
2. The ability to implement simple games and other elementary simulations.
3. A knowledge of the Model-Component design methodology.

6.2. Introduction

Of all the different categories of components, the Model is probably the most fun. A Model is used to encapsulate an Object Oriented Model of some useful object. Interactive puzzles and games can be created using Model components, as well as other more “serious” things. An example of such a model is the *Towers of Hanoi* component (abbreviated to just *Hanoi* in the following text) shown in Figure 6.1.

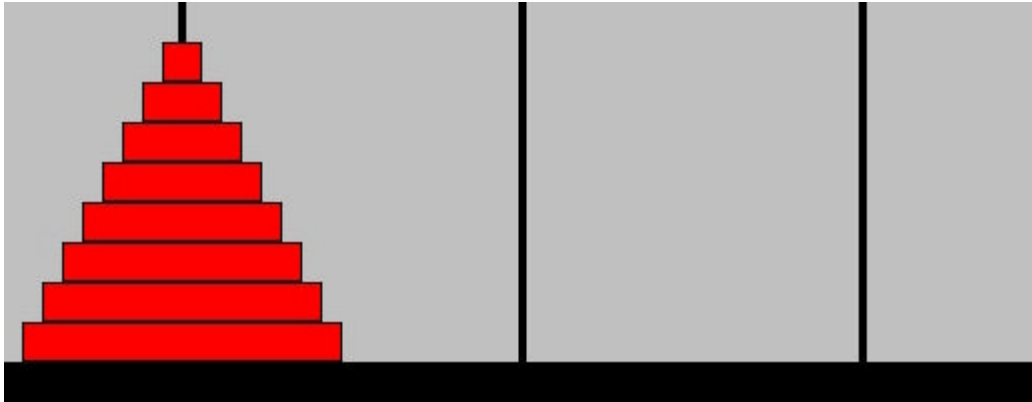


Figure 6.1. The Towers of Hanoi.

A Model component contains an internal object model that is manipulated using the methods of the component. Generally the object model is an integral part of the component, being designed along with the component, and permanently embedded in the component. When the Model component executes, the internal object model is already complete. This is opposed to an Editor component, which is used to construct the internal object model. The internal object model is not normally saved in permanent storage, but some variants of the Model component permit this.

Most Model components have a display that shows the internal state of the internal object model. In most cases, the user can interact with the internal object model by clicking on various parts of the display.

6.3. The Design Methodology

The design of a Model component proceeds in four stages, the design of the internal object model, the design of the display, the design of the component's manipulator methods, and the design of the user interaction. The design of the internal object model is the most important step. In effect, the internal object model *is* the component. The

remaining steps in the design merely provide a wrapper for the internal object model.

Properties are used both as model parameters and as display parameters. Model parameters are those that have some effect on the model itself. For example, in the Hanoi component, a model parameter might be the total number of disks in the puzzle. (In Figure 6.1, this parameter would be set to 8.) Display parameters affect only the display. They are a part of the component interface, but not a part of the model. An example of a display parameter for the Hanoi component might be the height of each disk. The properties of a Model component are designed both during the design of the internal object model, and during the design of the display.

Methods are used for manipulating the model. In the Hanoi component, methods will be used to move a disk from one peg to another, and to reset the game to its initial state. Methods are usually wrapper functions for the functions of the internal object model.

Events are used for user interaction. Typically they are used to report mouse clicks. It is generally necessary to analyze the position of the mouse cursor to determine what part of the model has been clicked. The click event of the Hanoi component will report which peg (if any) has been clicked. Events can also be used to report error conditions.

Although we have not done this in the Hanoi, we could use an event to report an attempt to place a large disk on top of a smaller one.

In the remainder of this chapter we will illustrate the design of a Model component using two games as examples. Model components can also be used for more serious purposes, but games provide a “fun” way to begin learning about component-level design.

6.4. *The Towers of Hanoi*

The display of the Hanoi component represents a board with three pegs, on which there are eight disks of different sizes. The disks are stacked pyramid-style on peg 1. The object of the puzzle is to move the stack of disks from peg 1 to peg 2, using peg 3 as a work peg. Disks must be moved one at a time without ever placing a large disk on top of a smaller one. The problem is known to be exponential in the number of disks.

The Hanoi control contains an object-oriented model of the peg-board and the disks. This model was developed without regard to the fact that it would eventually be used in a component, or more specifically, in an ActiveX control. Typical Object-Oriented-Design principles were used, although with somewhat less rigor than we would use for a more “serious” component. As the object model was being created, care was taken to make sure that any program using the model would have a sufficient number of public functions and variables to query the object state and perform the required operations on it.

The design of the model began with the class **CPeg** which incorporates all of the functionality required to model a peg. The class contains three data items, *Data*, *Slots*, and *Disks*. The *Data* item is a dynamic array of integers representing the size of each disk on the peg. The sizes 1-8 would be used for an eight-peg game. The item *Slots* contains the capacity of the peg, i.e. the maximum number of disks the peg can hold, while the item **Disks** contains the current number of disks on the peg. In the *Data* array, the size of the lowest disk is found in array item 0, with the higher indices indicating the disks higher on the peg. If there are k disks on the peg, their sizes will be found in array items 0

through $k-1$. The three items are protected, because changes to the items must be done in a coordinated way.

The **CPeg** class has four manipulator functions *AddDisk*, *RemoveDisk*, *Empty* and *Fill*. The *AddDisk* and *RemoveDisk* functions are used to move disks between pegs, while the *Empty* and *Fill* functions are used during initialization. The *Empty* function removes all disks from a peg, while the *Fill* function places all disks, in order, on the peg. The **CPeg** class also has two accessor functions, *DiskCount*, which returns the current number of disks on the peg, and *DiskSize* which returns the size of the disk in a particular position on the peg. The **CPeg** class also has a special constructor that is used to construct pegs of a specific size. The definition of the **CPeg** class is given in Figure 6.2.

The **CPeg** class is used as a part of the **CBoard** class. The **CBoard** class is used to encapsulate the functionality of the entire game board. The **CBoard** class contains two protected data items, *NoDisks*, which gives the number of disk capacity of each peg, and *Pg*, an array of three pegs. There is also a public data item, *Selected*, that indicates which peg, if any, is selected. Any value of *Selected* other than 0, 1, or 2 is used as a “no selection” indicator, so there is no need to make this item protected. The **CBoard** class has a special constructor that is used to create a board of a specific size. The default constructor creates a board with eight disks. There is an *Initialize* function that moves all disks to peg 1 and empties pegs 2 and 3. There are two manipulator functions, *MoveDisk* and *MoveStack*, and three accessor functions *DiskCount(void)*, *DiskCount(long)*, and *DiskSize*. The *MoveDisk* function is used to move a single disk from one peg to another, while the *MoveStack* function is used to move an entire stack of disks starting with the top disk. The *MoveStack* function does not necessarily move *all* disks on a peg. The

accessor functions are used to extract information about the disks on a peg, primarily so that the board can be drawn properly. Figure 6.3 gives the definition of the **CBoard** class.

```
class CPeg
{
protected:
    long * Data;
    long Slots;
    long Disks;

public:
    Peg();
    Peg(long NoDisks);
    virtual ~Peg();

    // initialization
    void Empty(void);
    void Fill(void);
    // movement
    long AddDisk(long Size);
    long RemoveDisk();
    // accessor functions
    long DiskSize(long DiskID);
    long DiskCount(void);
}
```

Figure 6.2. The CPeg Class Definition.

```
class CBoard
{
protected:
    long NoDisks;
    Peg * Pg[3];
public:
    long Selected;

public:
    Board();
    Board(long DiskCount);
    virtual ~Board();

    void Initialize();
    // movement
    long MoveStack(long From, long To, long Work, long Count);
    long MoveDisk(long From, long To);
    // accessors
    long DiskCount(void);
    long DiskCount(long PegID);
    long DiskSize(long PegID, long DiskID);
};
```

Figure 6.3. The CBoard Class Definition.

Figure 6.4 shows the implementation of the *AddDisk* and *RemoveDisk* functions. Note that the *Disks* array is treated as a formal stack. This makes sense because disks must be removed in the reverse order from which they were placed on the peg. The *RemoveDisk* function must check for the empty peg error, while the *AddDisk* function must check for peg overflow and stacking a large disk on top of a smaller one.

```
long CPeg::RemoveDisk()
{
    if (Disks > 0)
    {
        Disks--;
        return Data[Disks];
    }
    else
    {
        return -1;
    }
}

long CPeg::AddDisk(long Size)
{
    if (Disks >= Slots)
    {
        return 0; // programming error if we get here
    }
    if (Disks <= 0)
    {
        // first disk on the peg 0=bottom disk position
        Data[0] = Size;
        Disks = 1;
    }
    else
    {
        if (Size > Data[Disks-1])
        {
            return 0; // illegal move
        }
        Data[Disks] = Size;
        Disks++;
    }
    return 1;
}
```

Figure 6.4. The AddDisk and RemoveDisk Functions.

Figure 6.5 shows the Empty and Fill functions which are used by the constructors and by the *Initialize* function. Figure 6.6 shows one CBoard constructor, and demonstrates how the board is initialized.

```
void CPeg::Empty()
{
    Disks = 0;
}

void CPeg::Fill()
{
    for (long i=0 ; i<Slots ; i++)
    {
        Data[i] = Slots - i;
    }
    Disks = Slots;
}
```

Figure 6.5. The Empty and Fill Functions.

```
CBoard::CBoard(long DiskCount)
{
    NoDisks = DiskCount;
    Pg[0] = new Peg(NoDisks);
    Pg[1] = new Peg(NoDisks);
    Pg[2] = new Peg(NoDisks);
    Selected = -1;
    Pg[0]->Fill();
    Pg[1]->Empty();
    Pg[2]->Empty();
}
```

Figure 6.6. The CBoard Constructor.

Figure 6.7 shows the *MoveDisk* function of the **CBoard** class. This function calls *RemoveDisk* on the *From* peg, and *AddDisk* on the *To* peg. The rest of the code is error checking.

```
long CBoard::MoveDisk(long From, long To)
{
    if (From > 2 || From < 0 || To > 2 || To < 0)
    {
        return 0;
    }
    long Disk = Pg[From]->RemoveDisk();
    if (Disk == -1)
    {
        return 0;
    }
    long rv = Pg[To]->AddDisk(Disk);
    if (!rv)
    {
        Pg[From]->AddDisk(Disk);
    }
    return rv;
}
```

Figure 6.7. The MoveDisk Function.

The *MoveStack* function is given in Figure 6.8. This function moves an entire stack of disks from one peg to another. To move a stack of disks, the algorithm uses a source peg, a target peg and a work peg. To move k disks, $k-1$ disks are moved to the work peg, then the bottom disk is moved from the source peg to the target peg, and then the $k-1$ disks are moved from the work peg to the target peg. The $k-1$ disks are moved using a recursive call to *MoveStack*. If the stack is only one disk high, then a single disk move is used to move the stack. *MoveStack* checks for two errors, invalid peg numbers, and requested number of disks not available.

```

long Board::MoveStack(long From, long To, long Work, long Count)
{
    if (From > 2 || From < 0 || To > 2 ||
        To < 0 || Work > 2 || Work < 0 ||
        From == To || From == Work || To == Work)
    {
        return 0;
    }
    if (Count <= 0 || Count > Pg[From]->DiskCount())
    {
        return 0;
    }
    if (Count == 1)
    {
        MoveDisk(From,To);
    }
    else
    {
        MoveStack(From,Work,To,Count-1);
        MoveDisk(From,To);
        MoveStack(Work,To,From,Count-1);
    }
    return 1;
}

```

Figure 6.8. The MoveStack Function.

After the design of the internal object model was completed, the model was embedded in an ActiveX framework. This framework also contains the other three parts of the design, the drawing routine, the manipulator methods, and the user interaction functions. The board is implemented as a single data item, *MyBoard*, of the ActiveX support class.

The drawing routine is straightforward. The bottom of the board and the pegs are drawn as black rectangles, and the disks are drawn as red or blue rectangles with black borders. The pegs are drawn to be “one disk taller” than the maximum number of disks that can be placed on a peg. The distance between pegs is calculated to be half the width of the widest disk plus ten pixels. Ten pixels of space are left on each side of the board. The top of each peg will always touch the top of the control window. The height of each disk is 20 pixels, and the width is its size times 20. Therefore the disk of size 5 is 100

pixels wide. Each disk is drawn as a red rectangle with a black border. If a stack is selected, the top disk in the stack is drawn as a blue rectangle with a black border. We omit the details of the drawing routine, since much of this routine is Windows operating system functions that set drawing modes, background colors, and so forth. During this phase of the design we added a single property, `BackColor`, which is used to set the background color. The property design table for this property appears in

Property Design Table		
Name	Type	Function
BackColor	OLE_COLOR Default = White	Specifies whether the background color for the drawing. This color will fill all empty space in the drawing.

Figure 6.9. BackColor Property Description.

During the third phase of the design a collection of manipulator methods was added to permit external manipulation of the model. These methods are *Initialize*, *MoveDisk*, *Select*, *DeSelect*, *GetSelect*, *Solve*, and *Abort*. The formal method descriptions of these methods is given in Figure 6.10 through Figure 6.16. The simplest of these are the *MoveDisk* and *Initialize* methods, which are simply wrappers for the *MoveDisk* and *Initialize* functions of the **CBoard** class. Three methods, *Select*, *DeSelect*, and *GetSelect* were added to permit the manipulation of the selected peg. These methods use the *Selected* item of the board model, *MyBoard*. The *Select* function sets a specific peg as the selected peg, *DeSelect* clears the selection so no peg is selected, and *GetSelect* returns the current value of the selected peg, if any. *GetSelect* returns a value other than 0, 1 or 2 to indicate that no peg is selected. The final two manipulator methods are *Solve* and *Abort*. *Solve* is used to provide an animated solution of the puzzle, and *Abort* is used to

terminate the *Solve* method before completion. Permitting early termination of the *Solve* function is tricky because the *Solve* function must interrupt itself periodically to allow other portions of the program (such as buttons) to run. It is also necessary to slow the solution process down to provide a reasonable animation. Without some inserted delays, the process would complete too quickly to be visible. The animation is done by changing the state of the model, and then redrawing it. We do not show the intermediate steps as a disk moves from one peg to another.

Method Description			
Name	Initialize		
Return Value	Void		
Description	Moves all disks to peg 1, clears peg 2 and peg 3, redraws the window.		
Arguments	Name	Type	Description
Void			

Figure 6.10. Initialize Method Description.

Method Description			
Name	MoveDisk		
Return Value	Void		
Description	Moves the top disk from one peg to another. The parameters specify the source and target peg		
Arguments	Name	Type	Description
1	From	Long Integer	Source peg 0, 1, or 2.
2	To	Long Integer	Target peg 0, 1, or 2

Figure 6.11. MoveDisk Method Description.

Method Description			
Name	Select		
Return Value	Void		
Description	Selects a specific peg. Clears selection if selected peg is already selected. Does nothing if operand is invalid.		
Arguments	Name	Type	Description
1	Peg	Long Integer	The peg to be selected, 0, 1, or 2.

Figure 6.12. Select Method Description.

Method Description			
Name	DeSelect		
Return Value	Void		
Description	Clears the selected peg. No peg is selected after this call.		
Arguments	Name	Type	Description
Void			

Figure 6.13. DeSelect Method Description.

Method Description			
Name	GetSelect		
Return Value	Long Integer		
Description	Returns the number of the selected peg, or -1 if no peg selected.		
Arguments	Name	Type	Description
Void			

Figure 6.14. GetSelect Method Description.

Method Description			
Name	Solve		
Return Value	Void		
Description	Shows an animated solution to the puzzle.		
Arguments	Name	Type	Description
Void			

Figure 6.15. Hanoi Solve Method Description.

Method Description			
Name	Abort		
Return Value	Void		
Description	Aborts an animated solution of the puzzle.		
Arguments	Name	Type	Description
Void			

Figure 6.16. Hanoi Abort Method Description.

In creating the Solve routine, it was not possible to use the *MoveStack* function of the **CBoard** class, because this function does not (and *should* not) contain the operating system function calls used to redraw the model, delay the drawing, and interrupt the solution process to permit other portions of the program to run. Therefore it was necessary to implement an external *MoveStack* routine that contains all the functionality of the **CBoard** *MoveStack* function, plus the additional operating system calls. This function is called *LMoveStack*, and interacts with the internal object model using the *MoveDisk* function. Figure 6.17 gives the implementation of *LMoveStack*. This function uses the operating system function *CWnd::Invalidate* function to redraw the window. To solve the puzzle, it is necessary to call the *Initialize* function to place all disks on Peg 1, and then call *LMoveStack* (once) to move all disks from Peg 1 to Peg 2.

The *Pause* function delays the program the specified number of milliseconds (500=1/2 second), and also momentarily gives up control of the CPU to let other portions of the program run. The implementation of this function is given in Figure 6.18. This function does not actually relinquish the CPU, but instead finds all pending work for other parts of the program, and dispatches it to the proper program subroutine for processing. This is done through two Windows operating system calls, *PeekMessage*, and *DispatchMessage*. During this processing, it is possible for a button to be clicked and for the Abort method of the ActiveX control to be called. This will set the AbortSolve

variable to **true**. Once this variable is set to **true**, the solve routine will abort itself without completing its task.

Delays are implemented by waiting for an event that will never occur using the specified delay as a timeout value. This is done using the *WaitForSingleObject* operating system function.

```
void CHanoiCtl::LMoveStack(long From,long To,long Work,long Count)
{
    ... // Error checking code omitted for brevity.
    if (Count > 1)
    {
        LMoveStack(From,Work,To,Count-1);
        if (AbortSolve)
        {
            return;
        }
        MyBoard->MoveDisk(From,To);
        CWnd::Invalidate();
        Pause(500);
        if (AbortSolve)
        {
            return;
        }
        LMoveStack(Work,To,From,Count-1);
    }
    else
    {
        MyBoard->MoveDisk(From,To);
        CWnd::Invalidate();
        Pause(500);
    }
}
```

Figure 6.17. The LMoveStack Function.

```
void CHanoiCtl::Pause(long mSec)
{
    MSG msg;

    while (PeekMessage(&msg, NULL, NULL, NULL, PM_REMOVE))
    {
        DispatchMessage(&msg);
    }
    if (AbortSolve)
    {
        return;
    }
    WaitForSingleObject(THandle, mSec);
}
```

Figure 6.18. The Pause Function.

The final phase of the design is the design of the user interaction. We want the user to be able to solve the puzzle manually by clicking on pegs with the mouse. Ideally, we would like to permit the user to move a disk by dragging it from one peg to another, but this is relatively complicated, so we will settle for something simpler. What we will do is this. We will allow the user to select a peg, and then move the disk from the selected peg to another peg by clicking on the target peg. The selection part has already been implemented, so we need to concentrate only on the movement. For this we will use a *passive model* method of user interaction. In this type of user interaction, all changes to the internal object model must be performed using manipulator methods. User interactions, mouse clicks and keystrokes, are filtered through the control and passed to the user in the form of events. The control makes no changes to the model in response to user interactions. We will not use keystrokes in our control, but we will use mouse clicks. We will create a single event, *Click2*, which we will fire in response to the user pressing the left mouse button. (We would like to use the name *Click* for our event, but some containers don't respond properly if *Click* is implemented in a non-standard way.) Our *Click2* event has a single parameter that indicates which peg has been clicked. We don't

want it to be necessary for the user to click precisely on the peg, but want it to be possible for the user to click anywhere on the stack of disks for the peg. We will return 0, 1, or 2 depending on which peg has been clicked, or -1 if the click is outside the drawing. We use the operating system `CWnd::GetClientRect` to get the rectangle for the control's window.

```
void CHanoiCtl::OnLButtonDown(UINT nFlags, CPoint point)
{
    RECT MyPosRect;
    CWnd::GetClientRect(&MyPosRect);
    long ScrWidth = 60*MyBoard->DiskCount()+40;
    long ScrHite = 40+20*MyBoard->DiskCount();
    long x = point.x;
    long y = point.y;
    if (x<MyPosRect.left || x > MyPosRect.left+ScrWidth)
    {
        FireClick2(-1);
    }
    if (y<MyPosRect.top || y>MyPosRect.top+ScrHite)
    {
        FireClick2(-1);
    }
    if (x<MyPosRect.left+20*MyBoard->DiskCount()+15)
    {
        FireClick2(0);
    }
    if (x<MyPosRect.left+40*MyBoard->DiskCount()+25)
    {
        FireClick2(1);
    }
    FireClick2(2);
}
```

Figure 6.19. Firing the Click2 Event.

When creating a program for manually solving the Towers of Hanoi puzzle, virtually all of the work can be done in the `Click2` event handler. Figure 6.20 shows the Visual Basic code for implementing the manual solution of the puzzle. In this event handler, we first check to see if a peg is selected. If not, then we select the clicked peg. If there is a selected peg, but the clicked position is not a valid peg, then we deselect the selected peg.

If there is a selected peg, and the clicked position is a valid peg, then we deselect the selected peg and move the top disk from the formerly selected peg to the clicked peg.

```
Private Sub HanoiCtl1_Click2(ByVal Pos As Long)
    Dim SelStat As Long

    SelStat = HanoiCtl1.GetSelect
    If SelStat = -1 Then
        HanoiCtl1.Select Pos
    ElseIf Pos = -1 Then
        HanoiCtl1.DeSelect
    Else
        HanoiCtl1.DeSelect
        HanoiCtl1.MoveDisk SelStat, Pos
    End If
End Sub
```

Figure 6.20. Solving Hanoi Manually.

This completes the design of the Hanoi ActiveX control. The CD contains the complete code for this component.

6.5. The Pentomino Puzzle

Our second example of a Model component is the Pentomino Puzzle illustrated in Figure 6.21.

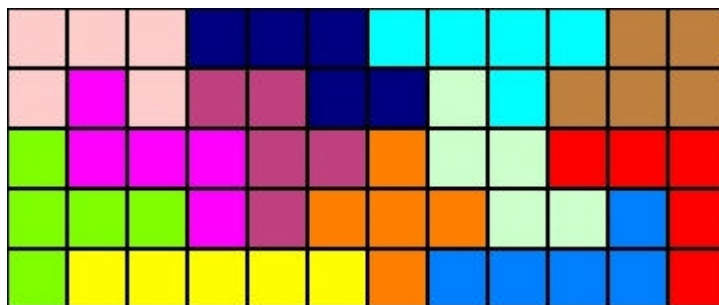


Figure 6.21. The Pentomino Puzzle.

The object of the Pentomino puzzle is to fit twelve odd-shaped pieces into a 5x12 rectangle. Each piece consists of five identical squares arranged so that each square shares at least one edge with another square. There are exactly twelve shapes that can be constructed in this fashion. Each shape is designated with a letter that more or less corresponds to the shape. A mechanical version of this puzzle with plastic pieces can be purchased at many game and toy stores. The mechanical version of this puzzle may or may not use a different color for each piece, but on the computer screen, it would be difficult or impossible to see the different shapes in the game board without using a different color for each shape.

In our design of the electronic Pentomino puzzle, we will follow the same methodology as we used for the Towers of Hanoi. We will start with the object model, then design the drawing routine for the model, then create manipulator methods, and finally add the user interface. As a first step in designing the object model, we will select a color to represent each piece. These colors, along with the shape letters are listed in Figure 6.22. These colors are identical to the ones shown in Figure 6.21, so this figure can be used as a reference to determine which letter goes with which shape.

Shape	Color
U	Pink
T	Green
S	Magenta
I	Yellow
Z	Dark Blue
K	Purple
X	Orange
Y	Blue-Green
W	Light-Green
L	Light-Blue
P	Brown
V	Red

Figure 6.22. Shape Color Assignments.

6.5.1. Internal Object Model Design

The object model for the Pentomino puzzle must contain two things, a 5x12 board and a collection of twelve shapes. Eventually we will need some additional objects to support the solution of the puzzle, but for now let's stick with what we know about. This object model is a bit more complex than that for the Towers of Hanoi, so we will use a slightly more rigorous methodology. To that end we will start with the object diagram shown in Figure 6.23.

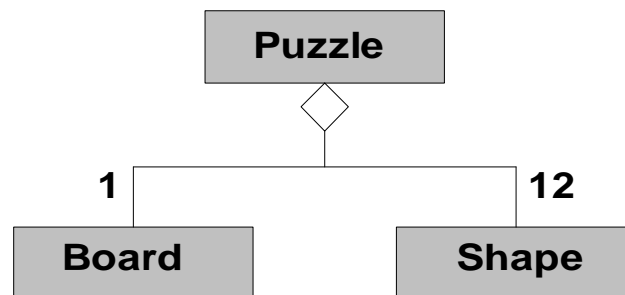


Figure 6.23. Preliminary Puzzle Class Diagram.

For each shape we need to specify how to draw the shape. This will be reasonably complicated, because each shape can be drawn in one of eight orientations, not all of which are different for each shape. The orientations are rotations of 0° , 90° , 180° , and 270° , and the mirror image of each rotation. Asymmetric shapes like K (purple) have eight orientations, X (orange) has one orientation, I (yellow) has two, and V (red) has four.

The simplest way to model the board is as a 5×12 array of integers. One method of managing the board would be to use the integers 1-12 to represent the twelve shapes, and the integer 0 to represent an empty square. However, since the codes we use are arbitrary, it is just as easy to use the shape color as an indicator. (Colors are represented using the low-order 24 bits of a long integer.) We will use the color White (0x00FFFFFF) to represent an empty square. When we draw the board, the color information we need to draw each square will already be in place in the proper position of the board. The information we keep for each shape must permit us to map the color of the shape into the proper position in the 5×12 array. Although we could enumerate the shapes automatically, and then enumerate each different orientation of each shape, it is much easier to generate this information manually and simply incorporate it into the source code of the program. Therefore, we will generate a constant list of twelve shapes, and for each shape we will generate a constant list of orientations. In this list we will list only those orientations that are different from one another.

In describing an orientation for a shape, it may seem necessary to represent all five squares. However this is not true. For each orientation we will designate one square as the primary square. This square will be implicit and will not be described in the model.

The model will contain information on the position of the other four squares with respect to the primary square. For the primary square, we will imagine the shape being embedded in a 5x5 array of squares. The primary square will be the top square in the leftmost column. Figure 6.24 illustrates the primary square for various orientations of the shape K. The selection of the primary square is motivated by the algorithm we will use to automatically solve the puzzle. This algorithm will be described fully below. In Figure 6.24, the primary square is designated by a cross through the square.

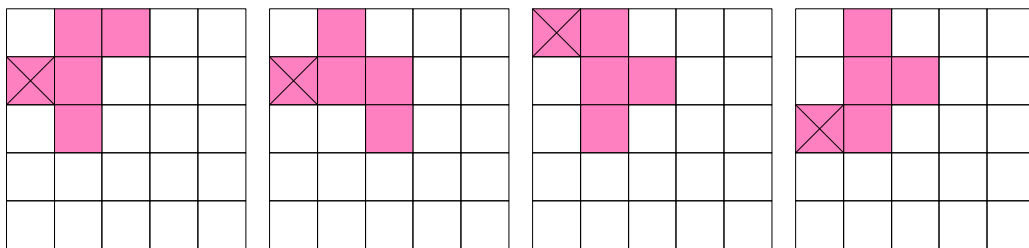


Figure 6.24. Some Orientations for Shape K.

Each orientation must contain four items of information, each of which will permit a non-primary square to be positioned with respect to the primary square. For each non-primary square, we will keep two integers. The first will indicate how many rows to the right one must move to locate the square, and the second will indicate how many rows up or down one must move to locate the square. Negative numbers indicate that the move is in the up direction. Thus for the first orientation of Figure 6.24 we must keep the following pairs of numbers: (1,-1), (1,0), (1,1), (2,-1). We require one class, which we will call the *move class*, to represent a pair of numbers, and another class to represent an orientation. The orientation class must contain an array of four moves. Each shape class must contain a variable sized array containing the orientations for that shape. In the shape

class, we must also have a counter that gives the size of the orientation array. The shape class should also contain the color of the shape. The final class diagram of our model is shown in Figure 6.25, and the C++ definitions of the classes are given in Figure 6.26. In keeping with the usual C++ practice we prefix each class name with the letter *C*. (We have also followed this practice with the Towers of Hanoi, and will follow this practice throughout this book.)

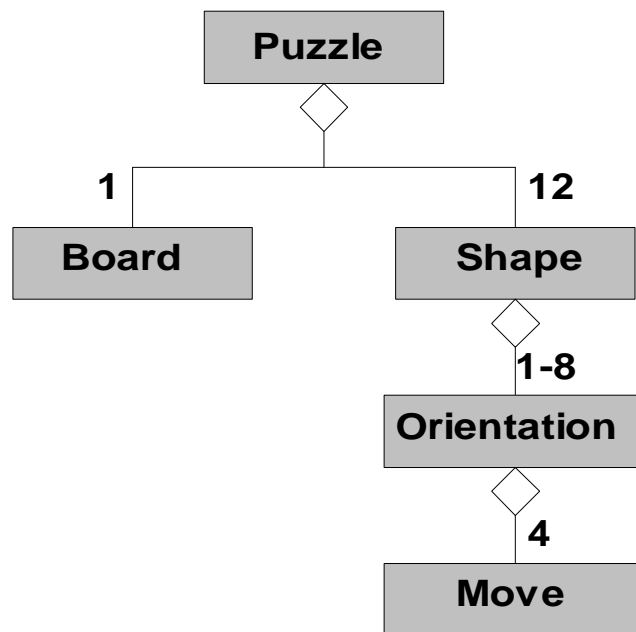


Figure 6.25 Final Puzzle Class Diagram.

<pre> class CSquare { public: long Over; long Down; }; </pre>	<pre> class COrientation { public: CSquare Square[4]; }; </pre>
<pre> class CShape { public: char Name; COLORREF Color; long OrientationCount; COrientation * Orientations; }; </pre>	<pre> class CPuzzle { public: CBoard Board; CShape Shapes[12]; }; </pre>
<pre> class CBoard { public: void Clear(); long AddShape(COrientation &Orientation, long StartRow, long StartCol, COLORREF Color); long RemoveShape(COrientation &Orientation, long StartRow, long StartCol, COLORREF Color); long NextEmpty(long StartRow, long StartCol, long &NewRow, long &NewCol); long NewBoard(long Rows, long Columns); COLORREF ** Data; long Columns; long Rows; }; </pre>	

Figure 6.26. Puzzle Class Definitions

For the most part, the class definitions should be no surprise. Note that the array of orientations in the CShape class is implemented as a pointer. This element will be dynamically allocated during the initialization phase of the component. Although we indicated that the board would have the dimensions 5x12, the actual implementation of the CBoard class allows the board-shape to change. To facilitate this, the Data element of the board is implemented as a double pointer. This allows it to be accessed as if it were a two-dimensional array, but also allows it to be reallocated. The default size of 5x12 will be set by the constructor of the class.

The class CBoard has been enhanced with four useful functions. The function *Clear* will erase the board to start a new puzzle. The functions *AddShape* and *RemoveShape*

will allow shapes to be added and deleted from the board. The function *NewBoard* will clear the board and change its size. The function *NextEmpty* will be used to locate an empty space in the board starting from the upper left and working down each column. This function will be used as part of the automatic solution of the puzzle.

Now that we have determined the final object model for the Pentomino puzzle, it is time to address the problem of automatically solving the puzzle. Although it is possible to solve this puzzle manually, it is devilishly difficult to do so. This is in spite of the fact that there are thousands of different solutions to the puzzle!

To solve the puzzle automatically, we first observe that in a valid solution, all 60 squares must be filled with some shape. We will use the strategy of searching the board for empty squares and attempting to fill each empty square with some shape. As shapes are inserted into the board, they will be marked as *used* so that no shape will be used twice. When an empty square is found, the algorithm will select an unused shape and attempt to fill the empty square with the shape. The algorithm will first place the primary square for the shape into the empty square and then attempt to place the other four squares based on the current orientation of the shape. This attempt can either succeed or fail. If the shape overlaps other shapes already in the board the placement fails. If the shape overlaps the edge of the board then the placement fails. Otherwise the placement succeeds. When a placement succeeds, the algorithm will record the details of the placement in a list of successful placements. A solution to the puzzle consists of a list of twelve successful placements.

As anyone who has attempted to solve this puzzle knows, there is more to finding a solution than simply selecting shapes in order and placing them in a rectangle. It is

possible to “go down the garden path” to a partial solution that will never lead to a complete solution. This is also true when solving the puzzle automatically. As with a manual solution, it is sometimes necessary to go back to some shape that has already been placed in the board and remove it. When an attempt to fill a square with a particular shape fails, the algorithm will first repeat the attempt with the next orientation of the shape. Once all orientations of a shape have been tried, the algorithm will proceed to the next shape and try the first orientation of that shape. When the list of unused shapes becomes exhausted without filling a square, the algorithm backs up to the last successful placement. That placement is undone, and the attempt is treated as a failed placement attempt. The algorithm will continue with the next orientation or shape for the previously successful attempt.

Those familiar with backtracking algorithms will recognize that this is a blind backtracking algorithm and is inherently exponential (or worse!). If there were a single unique solution to the puzzle this would probably make the algorithm intractable, however because there are actually thousands of valid solutions to the puzzle, the algorithm runs quickly.

To keep track of successful placement attempts, we need to record four items of information, the row and column of the square we were attempting to fill, the shape that we used to fill the square, and the orientation we used. We will provide a class to contain these four items of data, and will add an array of twelve of these objects to our **CPuzzle** class. We can also use this class to keep track of the current placement attempt. Figure 6.27 shows the enhancement of the class diagram to include a State class and Figure 6.28 shows the C++ definition of this class along with the required changes to the CPuzzle

class. We have added the *Solve* function, which will generate the automatic solution of the puzzle. The array, *ShapesUsed* will keep track of the shapes that have been used, and the variable *SuccessfulPlacements* will keep track of the current location in the *Placements* array.

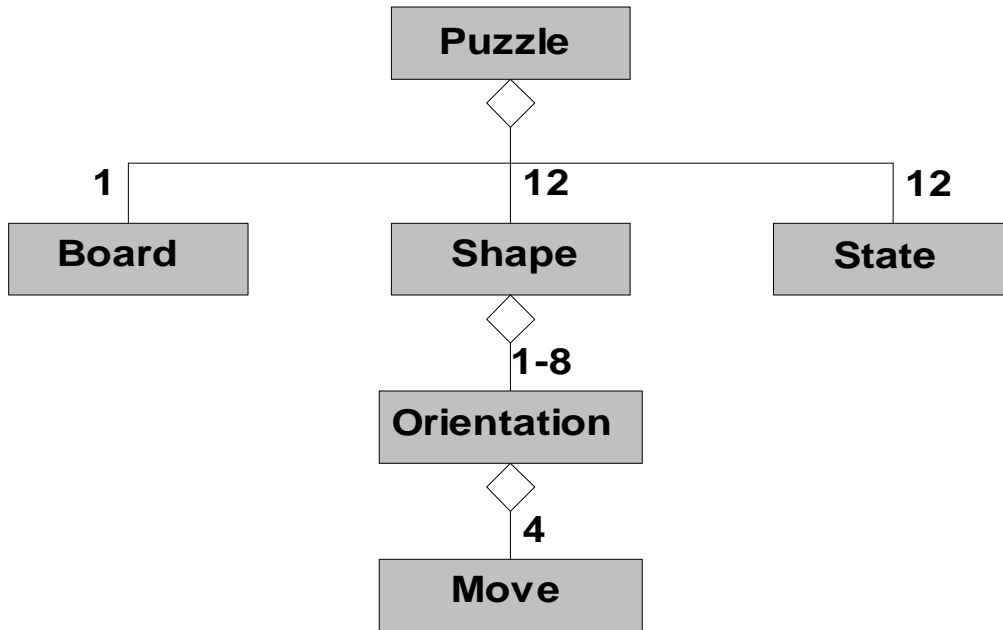


Figure 6.27. The New Final Class Diagram.

<pre> class CState { public: long Column; long Row; long Shape; long Orientation; }; </pre>	<pre> class CPuzzle { public: CBoard Board; CShape Shapes[12]; long ShapeUsed[12]; CState Placements[12]; long SuccessfulPlacements; void Solve(); }; </pre>
---	--

Figure 6.28. Enhancing the Pentomino Model.

Apart from the **CPuzzle** *Solve* routine, which we will present later, the most interesting functions in the Pentomino model are the *AddShape* and *RemoveShape* functions of the **CBoard** class. Figure 6.29 gives the code for the *AddShape* function. This function first checks each non-primary square to make sure it doesn't fall outside the board or overlap another shape. If the placement is successful it copies the shape color into each affected square of the board. If the placement is successful, the function returns 1, otherwise it returns zero.

Figure 6.30 shows the code for the *RemoveShape* function. Except for the color references, this routine is identical to the *AddShape* function. The function first tests to see that the shape is really where we claim it is, and then replaces the shape color with the color white (RGB(255,255,255) is white).

```
long CBoard::AddShape(COrientation &Orientation, long SRow,
    long SCol, COLORREF Color)
{
    ... // error checking code omitted for brevity
    for (long i=0 ; i<4 ; i++)
    {
        long NRow = SRow + Orientation.Square[i].Down;
        long NCol = SCol + Orientation.Square[i].Over;
        if (NRow < 0 || NRow >= Rows || NCol < 0 || NCol >= Columns)
        {
            return 0; // outside the board
        }
        if (Data[NRow][NCol] != RGB(255,255,255))
        {
            return 0; // overlaps another shape
        }
    }
    // successful, so change board colors
    Data[SRow][SCol] = Color;
    for (i=0 ; i<4 ; i++)
    {
        long NRow = SRow + Orientation.Square[i].Down;
        long NCol = SCol + Orientation.Square[i].Over;
        Data[NRow][NCol] = Color;
    }
    return 1;
}
```

Figure 6.29. The AddShape Function.

```

long CBoard::RemoveShape(COrientation &Orientation, long SRow,
    long SCol, COLORREF Color)
{
    ... // Error checking code omitted for brevity
    for (long i=0 ; i<4 ; i++)
    {
        long NRow = SRow + Shape.Square[i].Down;
        long NCol = SCol + Shape.Square[i].Over;
        if (NRow < 0 || NRow >= Rows || NCol < 0 || NCol >= Columns)
        {
            return 0; // Error! shape would be outside the board
        }
        if (Data[NRow][NCol] != Color)
        {
            return 0; // Error! Wrong shape in this position
        }
    }
    Data[SRow][SCol] = RGB(255,255,255);
    for (i=0 ; i<4 ; i++)
    {
        long NRow = SRow + Shape.Square[i].Down;
        long NCol = SCol + Shape.Square[i].Over;
        Data[NRow][NCol] = RGB(255,255,255);
    }
    return 1;
}

```

Figure 6.30. The RemoveShape Function.

This completes the design of the object model, now let us proceed to the next phase of the design.

6.5.2. Drawing-Routine Design

When constructing the drawing routine we must be concerned with two things, the appearance of the component, and user interaction with the component. These two aspects of the component should be kept in mind when designing the drawing routine.

For the Pentomino puzzle, we obviously want to draw the board that appears in Figure 6.21, but is this all? The object model has two other major parts, the shapes, and the solution state. We can easily dismiss drawing the solution state, because it contains no information that is not visually obvious in the drawing of the board. However, we may wish to arrange for drawing the individual shapes in addition to drawing the board. Let us

first focus on drawing the board. The routine for drawing the board should look something like the pseudo code of Figure 6.31. (The drawing routine is much more complicated than this, but the complexity is due to the large number of operating system calls that are required, not to any inherent complexity in the drawing process.)

```
for (i=0 ; i<Puzzle.Board.Rows ; i++)
{
  for (j=0 ; j<Puzzle.Board.Columns ; j++)
  {
    Draw Rectangle of Color Puzzle.Board.Data[i][j].
  }
}
```

Figure 6.31. The Pentomino Drawing Routine.

As Figure 6.31 illustrates, the drawing routine simply draws a bunch of rectangles. The only real problem is determining the size of the rectangle. There are two choices. We could automatically scale the board to fit in the existing control window, or we could use fixed-sized squares and require the user to create a control window of the proper size. If we decide to use a fixed size, it is a good idea to use a property value as the “fixed size.” This will allow the user to change the size of the squares to fit his or her needs. My preference is to use a fixed size and to specify the size with a property value. However, let us first begin by discussing how to scale the board into the existing window. First, we should set some minimum acceptable window size, and draw the window completely white if it is too small. Choosing a minimum window size is somewhat a matter of taste, so we will ignore that problem and proceed under the assumption that the current window size is acceptable.

First we must deal with the *aspect ratio* of the window. The *aspect ratio* of any rectangle is the ratio of the height and width. The aspect ratio of our board is 5::12.

Because it is unlikely that an arbitrary window will have this aspect ratio, we must be careful when scaling our board. We want the entire board to be visible, but we also want to fill as much of the window as possible. Figure 7 illustrates the desired appearance of the board for two different aspect ratios.

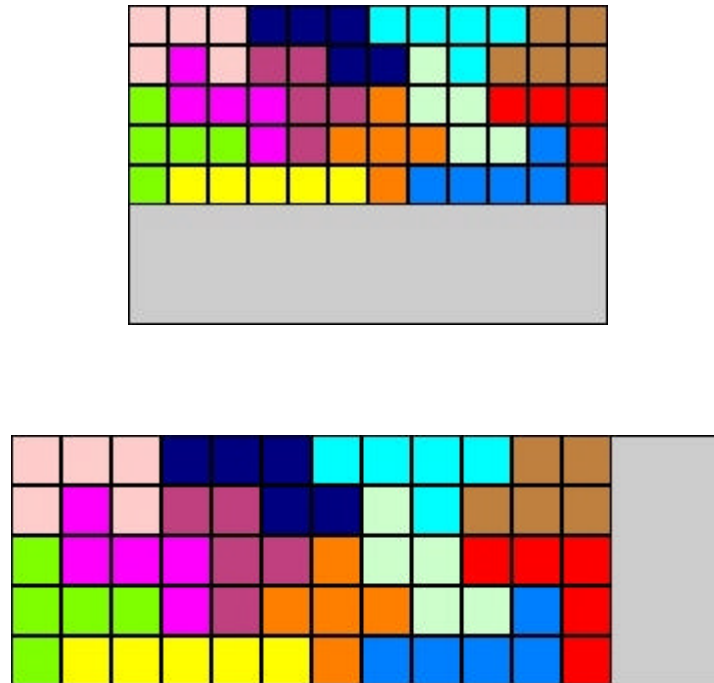


Figure 6.32. Scaling To Fit A Rectangle.

To find the appropriate square size for a window of height H and width W , we first compute $H/5$ and $W/12$. We must use integer division and ignore the remainder, because we cannot draw partial pixels. The value $H/5$ gives us the square size we must use to fit the board to the height of the window, and $W/12$ gives us the square size we must use to fit the board to the width of the window. To fit the board entirely inside the window we must choose the minimum of $H/5$ and $W/12$. The minimum of these two values will be used to draw each square.

To use fixed-sized squares, we first define a property named *SquareSize* of type long. We will provide a variable named *m_squareSize* to contain the value of the property. As with all components, we will define a support class to contain all “global” variables and functions for the component. The variable *m_squareSize* will be added to the support class of the component. Changes to the *SquareSize* property will be monitored. Values that are extraordinarily small or large will be rejected. If the new property value is acceptable, we will force a redraw of the component window.

In addition to drawing the board, we may also wish to draw the list of shapes. However, rather than drawing all shapes and orientations at once, we will provide properties to specify the shape and the orientation to be drawn. Since it is unlikely that the user will want to see both the board and the individual shapes in the same window, we will provide a *DrawMode* property that specifies whether the board or an individual shape is to be drawn. The *DrawMode* property will have one of two values, “**0 - Board**” and “**1 - Shape**”. We will use a support-class variable named *m_drawingMode* to hold the value of this property. The new drawing routine will look something like Figure 6.33.

```
if (m_DrawingMode == 0)
{
    // Board-Drawing Routine;
    ...
}
else
{
    // Shape-Drawing Routine;
    ...
}
```

Figure 6.33. Shape/Board Drawing Routine.

The shape-drawing routine will use the *SquareSize* property and three new properties. The properties *Shape* and *Orientation* will be used to specify the shape and orientation to

be drawn, while the *MarkPrimary* property will be used to distinguish the primary square of a shape from the other four squares.

This component has no need to process keyboard input, but we wish to allow the user to interact with the display using mouse-clicks. When designing user-interaction we wish to maintain the *passive model* implementation style, so we will avoid performing complex behavior in response to user input. Instead, we will pass the mouse-click information back to the container through events. We will use two different events, one for each drawing mode. When drawing mode is set to “**0 - Board**” we will issue the *Board* event in response to mouse-clicks, otherwise we will issue the *Piece* event. The *Board* event will have two parameters, *X* and *Y*, which will be two long integers identifying the square on the board that has been clicked by the user. (If the user clicks off the board we will not issue an event.) The *Piece* event will also have two parameters, *Shape* and *Orientation*, long integers that give the shape number and the orientation number of the shape currently being displayed. (We are jumping ahead a bit in our methodology at this point, but for complex models it is generally impossible to isolate all four phases of the design.)

Figure 6.34 contains the property design table for the properties we have created. We will add entries to this table as we proceed with the program-interface design.

Property Design Table		
Name	Type	Function
DrawMode	Enumerated Default = 0	The value zero will cause the board to be drawn, the value 1 will cause the shape selected by the Shape and Orientation properties to be drawn. Causes Redraw.
Shape	Long Integer Maximum = 11 Minimum = 0 Default = 0	When DrawMode is equal to 1, specifies the shape to be drawn. No function when Drawmode is zero. Causes redraw if DrawMode is equal to 1. Resets Orientation to zero.
Orientation	Long Integer Maximum = Var. Minimum = 0 Default = 0	When DrawMode is equal to 1, specifies the orientation of the shape to be drawn. Gets reset to zero when Shape changes. Causes redraw when DrawMode is equal to 1. Maximum value is one less than the number of orientations of the shape specified by the Shape property.
SquareSize	Long Integer Maximum = 100 Minimum = 4 Default = 20	Determines the width and height of the individual squares in both the board display and the individual shape display. Causes Redraw.
MarkPrimary	Boolean Default = True	When DrawMode is equal to 1, a true value will cause an x to be drawn through the primary square of a shape, a false value will cause all squares to be drawn the same.

Figure 6.34. Initial Property Definitions.

Figure 6.31 suggests that we are equipped to handle boards other than the standard 5x12. However, as yet we have not provided any means for the user or programmer to select any board size other than 5x12. Clearly it would be desirable to add some method of changing the size of the board. We could simply expose the *NewBoard* function of the **CBoard** object, but this would permit board size changes only at run time. Most programmers would probably prefer to size the board at design time. If we wish to give the programmer this capability, we must add properties that permit the size of the board to be changed. It would be nice if we could change both dimensions at the same time, because this would allow us to verify that the board contained 60 squares.

Unfortunately we have come to an impasse. The only thing available to the programmer at design time is properties, but properties can have only simple types. Changing both dimensions of the board simultaneously would require a structure containing two numbers. Since we are forced into a situation where we must permit one dimension to be changed at a time, we must be careful not to generate an error when the board has more or less than 60 squares. To facilitate the height and width changes we will add two properties, *BoardWidth* and *BoardHeight*. Figure 6.35 gives the property design table entries for these properties.

Property Design Table		
Name	Type	Function
BoardWidth	Long Integer Minimum = 3 Maximum = 20 Default = 12	Specifies the number of columns in the board. Causes redraw, and puzzle-reset.
BoardHeight	Long Integer Maximum = 3 Minimum = 20 Default = 5	Specifies the number of rows in the board. Causes redraw and puzzle-reset.

Figure 6.35. Board Property Definitions.

6.5.3. The Manipulator Methods and User Interaction

We are finally ready to begin designing our manipulator methods. As we proceed with this, it will become clear that, in the beginning, we didn't have a clear idea of how the component would be used. This may require us to go back and make a few changes the object model and in the drawing routine. For complex models, this is to be expected. Although many design methodologies suggest that everything can and should be thought out beforehand, this is simply impossible. However, if we have made good decisions in the beginning, we should be able to incorporate whatever changes are required with a

minimum of pain. The true mark of a well designed object model is not its initial perfection, but the ease with which it can be modified to perform new tasks.

The only manipulator method that we have discussed so far is the *Solve* method, which is currently implemented as the *Solve* function of the *CPuzzle* class. Hopefully we will be able to simply provide a wrapper for this routine. To do this, we provide a method named *Solve*, with the initial intention of simply calling *Puzzle.Solve()*, and forcing a redraw of the control window. However, as we are completing the design document for this method (Figure 6.36), a new idea occurs to us. Why not let the user watch the solution develop by redrawing the control window whenever a shape is added or removed from the board? The user might not always want to watch the solution, so we could provide a new property, *ShowWork*, to permit the user to choose whether to watch the solution of the puzzle develop. Figure 6.36 gives the Method Description of for the *Solve* method, and Figure 6.37 gives the property design table entry for the *ShowWork* property.

Unfortunately, adding this new feature will force us to develop a new version of the *Solve* routine that is external to the **CPuzzle** class. The object-model of the puzzle does not know about (and *should* not know about) the component's drawing routine. This is the same problem we encountered with the Towers of Hanoi puzzle, and we will have to solve it in much the same way.

Because watching the solution develop might become long and tedious, it might be a good idea to have some means of aborting the solution. We will add an *Abort* method for this purpose, the description of which is given in Figure 6.38. When aborting a solution, we will not clear the board, but will leave the partial solution in place.

Method Description			
Name	Solve		
Return Value	Void		
Description	Clears the board, marks all shapes as unused, and finds a solution using a blind-backtracking algorithm. If the ShowWork property is set to TRUE , the control window will be redrawn whenever a piece is successfully placed, or successfully removed.		
Arguments	Name	Type	Description
Void			

Figure 6.36. Pentomino Solve Method Description.

Property Design Table		
Name	Type	Function
ShowWork	BOOL Default = TRUE	During automatic solution of the puzzle, specifies whether each successful shape placement or removal will cause a redraw of the board.

Figure 6.37. ShowWork Property Definition.

Method Description			
Name	Abort		
Return Value	Void		
Description	Aborts the search for a solution. The board is not cleared by this method. It is redrawn with the partial solution in place. Does nothing if solution is not in progress.		
Arguments	Name	Type	Description
Void			

Figure 6.38. Abort Method Description.

The next idea that occurs to us is providing some means for allowing the user to solve the puzzle manually. Even though the solution is difficult, there are people who enjoy difficult puzzles, and we should try to accommodate them. Obviously two methods are needed, one to add a shape to the board, and another to remove a shape. (We will not allow users to drag shapes.) Before we can specify the parameters of the *AddShape* and

RemoveShape functions, we must have a clear picture of how our component will be used, and what data will be available when each function is called. We also need to be concerned about the object model, since the **CPuzzle** class has no *AddShape* and *RemoveShape* functions. Because we have made everything public we could “reach into” the model and call the **CBoard** functions directly, but doing this is usually a mark of bad design. Reaching into a model makes the model difficult to understand and even more difficult to modify.

We will leave the correction of the model to the user, and focus instead on how we expect the model to be used during a manual solution of the puzzle.

When adding a shape, the most likely scenario is that the program will somehow select a shape and an orientation, and pass these parameters to the *AddShape* function. Since it is “cruel and unusual punishment” to expect the user to provide shape and orientation numbers directly, we must provide some method of obtaining these automatically. We have already designed a drawing mode that will permit individual shapes and orientations to be displayed, and have designed an event that will be fired when the shape is clicked. We can use these features to obtain our shape and orientation numbers. We will ask the programmer to provide a second copy of the Pentomino component that will be used to display shapes and orientations. The user should be able to manipulate this display in some fashion to go through the available shapes and orientations, and should be able to click on this component to select an orientation of a shape. Once an orientation is selected, the user can click on the square of the board where the shape should be placed. The component displaying the board will then issue an event containing the row and column numbers of the clicked square. The data that will be

available for placing a shape will be the shape number, the orientation number, and the row number and the column number. This is sufficient to place a shape in the board.

Figure 6.39 gives the design of the *AddShape* method based on these assumptions.

Method Description			
Name	AddShape		
Return Value	Long Integer	Error Code if placement is not successful.	
Description	Adds a new shape to the board. The arguments Shape and Orientation determine the shape to be added. The shape must be currently unused. Adding the shape will cause it to be marked as used. The Shape and Orientation arguments must be valid. X and Y determine the position of the primary square of the shape orientation. X and Y must be within the board. The new shape must not overlap any existing shapes or extend outside the board. The return value indicates whether adding the shape was successful. This method will add an entry to the list of successful placements.		
Arguments	Name	Type	Description
	Shape	Long Integer	Shape number of the shape to be added.
	Orientation	Long Integer	Orientation number of the shape-orientation to be added.
	X	Long Integer	Column number of the square where the shape is to be added. (Zero based)
	Y	Long Integer	Row number of the square where the shape is to be added.

Figure 6.39. AddShape Method Description.

Although the user will specify a shape and an orientation when adding a shape, it would be easiest if the user could remove a shape just by clicking on it. The information available for removing a shape would be just the row and column numbers of the clicked square. This is sufficient, because the color of the board-square can be used to look up the shape, then the shape can be looked up in the list of successful placements to find the orientation. Figure 6.40 gives the formal description of the *RemoveShape* method based on these assumptions.

Method Description			
Name	RemoveShape		
Return Value	Long Integer	Error code if removal is not successful.	
Description	Removes an existing shape from the board. X and Y must be inside the board, and must point to a non-white square. The color will be looked up in the list of shapes, then the shape number will be looked up in the list of successful placements. The successful placement will be removed, and the shape will be removed from the board.		
Arguments	Name	Type	Description
	X	Long Integer	Column number of the square where the shape is to be added. (Zero based)
	Y	Long Integer	Row number of the square where the shape is to be added.

Figure 6.40. RemoveShape Method Description.

The possibility of adding shapes to the diagram manually raises several questions with regard to the automatic solution of the puzzle. First, when solving the puzzle automatically, do we want to clear out all manually placed shapes and start fresh, or do we wish to attempt to build the solution around the manually placed shapes? As with most questions like this, the best answer is to find some way to do both. Let us assume that the method we have already designed, *Solve*, will clear the board before starting. Next, we will create a new method, *Continue*, which will solve the puzzle starting with any manually placed shapes. While we're at it we might as well add a *Clear* method that allows a user to erase the board and start over.

There are still two more problems we need to address. First, what happens if there is no solution with the manually placed shapes in their current positions? Do we include the manually placed shapes in the backtracking, or do we stop and report that there is no solution? Second, what happens if the *Continue* method is called on an aborted solution or on a completely solved board? (Recall that we do not clear the board when the

automatic search for a solution is aborted. Instead we display the board with some shapes already in place.)

We will allow the user to determine what happens when no solution can be found for a given manual placement. We will add a property, *RemoveManualOK*, which will be used to indicate whether manually placed shapes should be included in the backtracking process. The *Continue* method will resume a partially completed solution without clearing the board. If the method is called on a complete solution, the last placement will be removed and treated as unsuccessful. The solution will then continue from this point as with a partially completed solution. This allows a series of different solutions to be obtained by repeatedly calling the *Continue* method. Figure 6.41, Figure 6.42, and Figure 6.43 contain the formal descriptions of the new properties and methods.

Property Design Table		
Name	Type	Function
RemoveManualOK	Boolean Default = False	Specifies whether the Solve and Continue methods are permitted to remove manually placed shapes during the backtracking process. Default is to prevent the automatic removal of manually placed shapes.

Figure 6.41. RemoveManualOK Property Description.

Method Description			
Name	Continue		
Return Value	Void		
Description	<p>Attempts to find a solution to the puzzle using any manually placed shapes as a starting point. The same blind-backtracking algorithm as the Solve method is used to find a solution.</p> <p>A partial solution will be continued from the point where it was aborted.</p> <p>The last placement of a full solution will be backed out and treated as unsuccessful. The backtracking will then proceed from that point</p> <p>If the ShowWork property is set to TRUE, the control window will be redrawn whenever a piece is successfully placed, or successfully removed.</p>		
Arguments	Name	Type	Description
Void			

Figure 6.42. Continue Method Description.

Method Description			
Name	Clear		
Return Value	Void		
Description	<p>Remove all shapes from the board, regardless of how they were placed, and display an empty board.</p>		
Arguments	Name	Type	Description
Void			

Figure 6.43. Clear Method Description.

Keeping track of a manual solution requires still more modifications to the object model. At the very least, the **CState** class, which is used to keep track of placed objects, must be modified to indicate whether a placement was manual or automatic. There are many acceptable solutions to the problem, one of which can be found on the CD ROM. We will omit the details of this here, because handling manual placements is not substantially different from handling automatic placements.

6.5.4. The Solve Routine

It would be somewhat of a letdown to conclude this section without showing the routine that automatically finds the solution of the puzzle. Unfortunately, the routine is complicated, and there is no way to simplify it. The routine is shown in Figure 6.44. The basic plan of the routine is to find the first empty square starting from the upper left and working down each column. Then the routine attempts to fill the empty square with an unused shape. Each unused shape is tried, one by one, until they've all been tried, or until one fits. Each orientation of each shape is tried one by one.

The main loop is used for trial placements. If the trial succeeds, the loop continues with the next empty square and next unused shape. If it fails, the next orientation is tried. If there are no more orientations, the next unused shape is tried. If there are no more unused shapes, the routine backs up to the most recent successful placement and removes it. It then tries the next orientation for *that* shape. This may necessitate backing up even further. If the routine attempts to back up past the first successful placement, then there is no solution.

A modified version of this routine is used for the *Continue* method.

```

ClearBoard( );
// Set The Number of Successful Placements to Zero;
SuccessfulPlacements = 0;
for (long i=0 ; i<12 ; i++)
{
    ShapeUsed[i] = false;
}
// Initialize Placements[0] with the first placement attempt,
Placements[0]->Shape=0;
Placements[0]->Orientation=0;
Placements[0]->Column=0;
Placements[0]->Row=0;
DrawBoard( );
while (SuccessfulPlacements < 12)
{
    // At this point, the next placement attempt is located in
    // Placements[SuccessfulPlacements] For convenience, we will
    // retrieve Row, Column, Shape and Orientation from this object.
    CurrentShape = Placements[SuccessfulPlacements]->Shape;
    Orientation = Placements[SuccessfulPlacements]->Orientation;
    X = Placements[SuccessfulPlacements]->Column;
    Y = Placements[SuccessfulPlacements]->Row;
    Success = Board.AddShape(CurrentShape,Orientation,X,Y);
    if (Success)
    {
        // Mark the current shape as Used.
        ShapeUsed[CurrentShape] = true;
        SuccessfulPlacements ++ ;
        if (SuccessfulPlacements >= 12)
        {
            break; // exit the while statement;
        }
        // Set up next placement attempt. Find Next empty square,
        // starting with the square just filled
        Board.NextEmpty(X,Y,X,Y);
        // Find The Next unused shape;
        CurrentShape = FindFirstUnusedShape( );
        // If CurrentShape >= 12 there is a program error
        // and we should exit
        // Initialize Placements[SuccessfulPlacements] with
        // the proper data
        Placements[SuccessfulPlacements]->Shape=CurrentShape;
        Placements[SuccessfulPlacements]->Orientation=0;
        Placements[SuccessfulPlacements]->Column=X;
        Placements[SuccessfulPlacements]->Row=Y;
    }
    else
    {
        // We will try to create a new placement attempt,
        // backing up if necessary
        RetryPointFound = false;
        // keep backing up until we find something we haven't tried.
        while (!RetryPointFound)
        {
            Placements[SuccessfulPlacements]->Orientation++;
            CurrentOrientation =
                Placements[SuccessfulPlacements]->Orientation;

```

```

CurrentShape = Placements[SuccessfulPlacements]->Shape;
// First, try the next orientation.
if (CurrentOrientation < CurrentShape->OrientationCount)
{
    RetryPointFound = true;
}
else
{
    // If all orientations of the current shape
    // have been exhausted we next try the next
    // unused shape
    CurrentShape = GetNextUnusedShape(CurrentShape);
    if (CurrentShape < 12)
    {
        RetryPointFound = true
    }
    else
    {
        // if all shapes have been exhausted, then back up
        SuccessfulPlacements( );
        if (SuccessfulPlacements < 0)
        {
            // There is no solution exit inner loop
            break; // before retrying with zero
        }
        // Remove The Current Shape From The Board.
        CurrentShape =
Placements[SuccessfulPlacements]->Shape;
        Orientation =
Placements[SuccessfulPlacements]->Orientation;
        X = Placements[SuccessfulPlacements]->Column;
        Y = Placements[SuccessfulPlacements]->Row;
        Board.RemoveShape(CurrentShape,Orientation,X,Y);
        ShapeUsed[CurrentShape] = false;
    }
}
if (SuccessfulPlacements < 0)
{
    // no solution exit main loop
    break;
}
}
// Redraw The Board if ShowWork is true.
if (m_showWork)
{
    DrawBoard( );
}
// Abort if user has requested it.
if (UserAbort)
{
    break;
}
}
DrawBoard( );

```

Figure 6.44. The Pentomino Solve Routine.

6.6. Variant Forms of Models

Most Model components follow the pattern developed in the preceding two sections. There is a fixed object model, which is visible, and is manipulated by manipulator methods. There are Model components that do not conform to this pattern. Some Model components have non-fixed object models. Imagine an enhancement to the Pentomino component that permits users to save a partially completed puzzle and come back to it several days later. This is an example of a non-fixed model. Although saving and restoring states is normally associated more with Editor components than with Model components, the Model component does not permit items to be added to or deleted from the internal model. The only additional function that is permitted is saving and restoring the internal state of the model. The state of the model can be obtained either from fixed storage, usually a file of some sort, or through properties and methods. (See the discussion of semi-persistent objects in Chapter 20.)

Another type of non-fixed object model is a model that is *treated* as fixed by the Model component, but is actually modifiable by other components. An example of such a component is a simulator component that was created as part of the FHDL circuit design package. This component is capable of simulating a circuit whose description is passed to it as a semi-persistent object. The simulator stimulates the model with input data, and produces output data, but the model itself remains unchanged throughout the process. The entire model can be replaced at any time. Other components can be used to edit the model but the simulator component cannot be used for this purpose.

The simulator component is also an example of a Model component that has no visible display. The simulator produces data that is useful internally in the program and

which can be displayed by other components, but it has no display capabilities of its own. Needless to say, invisible Models are a specialized type of component that perform complex functions that are internal to a program, or produce data that must be massaged in some way before it is understandable to a user.

The non-fixed object model, and the invisible Model are important variations on the general theme of Model components.

6.7. Conclusion

Although a complex program may require many different objects, a Model control should be designed to represent one and only one object. If many objects of the same type are required in a program, each object should be represented by a single instance of the component. It is important to note that providing a component wrapper for an object adds a significant amount of overhead to an object. Because of this, the Model should be used to represent large-scale objects with significant and complex behavior.

The first step in designing a Model is to determine, in broad terms, how the internal object model is to be handled. The first thing we need to determine is whether the model will be visible or invisible. Many useful Models do not require a visible display or any user interaction.

Once we determine whether the model will be visible, we then must determine the source of the model. Specifically, will the object-model be permanently embedded in the component, as in the *Hanoi* and *Pentomino* controls, or will it be possible to switch between many different object-models. If it is not possible to switch between different object models, will it be possible to save and restore the state of the model? If *is* possible to switch between object-models, where will the component obtain them?

In any case, supporting multiple object-models is significantly more difficult than using a single embedded object model. Whenever possible, a single embedded object-model should be used.

Once we have determined the visibility and the object-source, we proceed to Object Oriented Design of the object model. Any acceptable object-oriented methodology can be used to design the object model, but for complex models it should include object diagrams and a significant amount of planning. Insofar as it is possible, the object oriented design of the object model should ignore the fact that the model will eventually be embedded in a component.

The design methodology for a Model has three steps that follow the design of the object model. The first of these is the design of the visible interface. Specifically, a drawing routine must be developed for the internal object model. If it will be necessary to redraw the internal object model frequently, the drawing routine must be written in such a way as to permit frequent redraws. In some cases it is necessary to begin thinking about interaction with the user when the drawing routine is designed. If the internal object model has several different views, it will be necessary to add properties and methods for switching between different views. Any time properties or methods are added to the component, it is necessary to have a clear idea of how the user will interact with the model to provide the appropriate values for these properties and methods.

If the component has a visible interface, it will usually be necessary to add a few drawing-parameter properties. Such properties are used to control scaling, background colors, borders, 3D shadowing, and other incidental aspects of the drawing. Before adding any drawing parameter, it is necessary to have a precise idea of exactly how that

parameter will be used. For example, if a component is designed to blend in with its background, it is unlikely that a border-drawing parameter would ever be used. If a full drawing of the model is too large to fit in a window of reasonable size, then it will be necessary to add scrolling capabilities to the window.

The next step in the design of a Model component is the creation of the programming interface. In most cases the programming interface will consist of a set of manipulator methods that are used to manipulate the model. During this phase of the design, it is necessary to concentrate exclusively on how the component will be used. One should not focus on a particular environment, unless that environment is known to cause special difficulties. One should go through the exercise of designing a program using the component so that one can visualize where the difficulties will lie. Very often it is will be necessary to go back and redo parts of the internal object model and the drawing routine to make them conform the programming interface.

The final step is the design of the user interaction. This step can be integrated with the design of the drawing routine and with the design of the programming interface. Most user actions will take the form of mouse-clicks, which can be associated with various parts of the visible model. Keyboard input is also possible, but for Model components, such input is rare. When designing the user interaction, it is best to adhere to the *passive model* style. In this style, all complex actions in the component are triggered by explicit method-calls. Mouse-clicks do not cause changes in the model, but are simply passed through to the container as events. This style of design permits the container to associate complex actions with certain mouse-clicks, but does not compel it to do so. The mouse-

clicks may be used in whatever manner the container chooses, permitting the component to be used in more versatile ways.

The Model component is a versatile and fun way to become familiar with component development, because it can be used to create interactive games and puzzles. Despite this, the Model component is also an important part of “serious” programming.

6.8. Exercises

1. Design a Model component for the Tic-Tac-Toe game, and write a Visual Basic program using the Model.
2. Design a Model component for playing the game of checkers and use it in a Visual Basic program.
3. Correct the problems in the object model of the Pentomino puzzle by changing the appropriate data items from public to protected and by adding the appropriate functions to the **CPuzzle** class.
4. Make the appropriate changes to the Hanoi component to permit saving and restoring the state of the puzzle in a file.
5. Make the appropriate changes in the Hanoi component to permit the user to change the number of disks. Limit the number of disks to the range 2 through 20.
6. Create a model component for your favorite game or puzzle. Create a web page to host your component and make your web page available on the Internet. (Send me E-Mail and I'll try it out.)