From:
Dan Olsen "Developing
User Interfaces" 1998,
Morgan Kaufmann Publishers

5

Basic Interaction

In the preceding chapters, we have discussed the basic concepts of event handling and graphical display. In this chapter, we will assemble these basic pieces together to construct an interactive application.

5.1 Introduction to Basic Interaction

The example we are going to use is shown in Figure 5-1. This application lays out simple logic schematics. The application has two windows, one for showing the circuit layout and connectivity and the other for showing a list of chips and their names.

The window on the left is the circuit view and has a menu of two modes on its far left. If the chip icon is selected, then the user can create new chips by clicking on any empty space in the chip view area. The chip will be added to the part list window without a name. The user can select a chip by clicking on it in the circuit view or in the part list view, in which case it will be highlighted in both views. If the DEL icon is selected, the currently selected chip is deleted from both views. The user can move a chip by pressing the mouse down on the chip and then dragging the mouse to a new location. The chip will be moved and all of the wires will stay connected.

When the wire icon is selected, the user can create new wires by pressing the mouse down on one of the chip connectors and dragging the mouse to another chip connector. This will create a wire between those two connectors. If the user clicks on an existing wire, it will become the selected wire. If the user then clicks on DEL, the wire will be deleted.

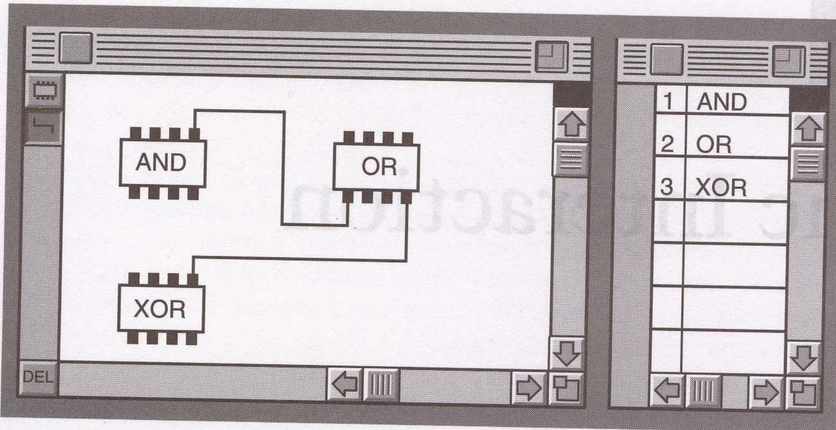

Figure 5-1 Logic schematic application

When a user selects a chip in the part list view, a character insertion point in the chip name is also selected. The user can then edit the chip name using the keyboard. The name of the chip will be updated both in the part list view and in the chip layout view.

This is a relatively simple application but it will illustrate several important concepts. In particular, we can look at how functional models are translated into model implementations. We can also look at architectures for handling multiple cooperating views. Finally, we can look at how the model and the actual user interface cooperate.

5.1.1 Functional Model

Having sketched the behavior of our example application, we need to define the functional model for the application. Since the purpose of this model is to demonstrate interactive architecture rather than to serve real users, we will skip the task analysis steps; in practice, this should never be done. There are three classes of objects in our functional model; they are Circuits, Chips, and Wires.

Circuit

A Circuit object captures the semantics of both the circuit view and the parts list view. The Circuit is the object being manipulated by both views. It is important to capture the concept of a circuit as an object rather than as a set of global variables. In any good application, it should be possible to view and edit

multiple circuits at the same time. This means that there will be multiple circuit objects. A set of global variables cannot handle this concept.

The data representation of a Circuit consists of the following:

`Chips`
: *An array of chip objects.*

`Wires`
: *An array of wire objects.*

`SelectedChip`
: *The index into Chips for the currently selected chip. This is –1 if there is no selected chip.*

`SelectedWire`
: *The index into Wires for the currently selected wire. This is –1 if there is no selected wire.*

There are a variety of methods on the class Circuit that provide the semantic behavior for our application, as follows:

`AddChip(CenterPoint)`
: *Add a new chip to the circuit at the specified location.*

`AddWire(Chip1, Connector1, Chip2, Connector2)`
: *Add a wire to the circuit that connects Chip1's Connector1 to Chip2's Connector 2.*

`SelectChip(ChipNum)`
: *Make the chip with the specified index the currently selected chip.*

`MoveChip(ChipNum, NewCenterPoint)`
: *Move the specified chip to the new location.*

`ChangeChipName(ChipNum, NewName)`
: *Change the chip's name.*

`DeleteChip(ChipNum)`
: *Delete the specified chip.*

`SelectWire(WireNum)`
: *Select the specified wire.*

`DeleteWire(WireNum)`
: *Delete the specified wire.*

These are the methods that the views can use to change circuits. These methods define the functionality of our application.

Chip

A Chip is a simple object that consists of the following:

`CenterPoint`
: *Center of the chip in the layout.*

`Name`
: *Name of the chip.*

In this simple application, the class Chip has no methods of its own. The entire functional behavior is captured in the Circuit class. In general, this would not be true. Circuits would consist of a variety of classes of circuit objects, each of which would have its own behavior. We will discuss more complex models in later chapters when we have more powerful geometric and architectural tools to handle them.

Wire

Wires are also quite simple and contain only their relevant data, as follows:

`Chip1`
: *Chip index to which the wired is connected.*

`Connector1`
: *Connector index in Chip1 to which the wire is connected. All Chips have exactly 8 connectors.*

`Chip2`
: *Chip index for the other end of the wire.*

`Connector2`
: *Connector index from Chip2 for the other end of the wire.*

5.2 Model-View-Controller Architecture

The Smalltalk system was developed as a language and an environment for building interactive applications.[1] As part of that development, an architecture for interactive applications was designed. This object-oriented approach was called the model-view-controller (MVC) architecture.[2] A schematic of this architecture is shown in Figure 5-2.

The *model* is the information that the application is trying to manipulate. This is the data representation of the real-world objects in which the user is interested. In our logic diagrams, the model would consist of the Circuit, Chip, and Wire classes.

The *view* implements a visual display of the model. In our application, there are two views, the circuit view and the part list view. Anytime the

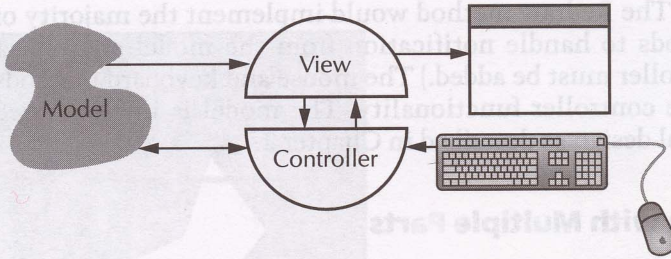

Figure 5-2 Model-view-controller

model is changed, each view of that model must be notified so that it can change the visual presentation of the model on the screen. A region of the screen that is no longer consistent with the model information is called *damaged.* When notified of a change, the view will identify the changed parts of the display and report those regions as damaged to the windowing system. In some systems, such regions are called invalid or out of date. In this text, we will use the term damaged. Reporting of damaged regions is fundamental to maintaining views on the screen.

A model, like ours, may have multiple views. In such a case, all views must be notified of the changes and the windowing system will collect them all. Later, when the main event loop looks for a new event to process, there will be redraw events waiting for any views that were affected by damage reporting and by any windowing operations. Each view must redraw the damaged areas based on information in the model. In addition to drawing the display, a view is also the location for all display geometry as will be discussed later.

The *controller* receives all of the input events from the user and decides what they mean and what should be done. In the circuit view of our example, the controller would receive a mouse-down event and must determine from the currently selected menu item whether wires or chips are to be manipulated. The controller must communicate with the view to determine what objects are being selected. For example, since the circuit view is responsible for positioning all of the chips in the window, the controller must be able to pass a mouse point to the view to determine if that mouse point is over a chip, a wire, or in empty space. Once the controller has all of the information that it needs, it will make calls on the objects in the model to make the appropriate changes. These calls by the controller on the model will cause the model to notify the views, and the displays will be updated.

Because the functionality of the controller and the view are so tightly intertwined and also because controllers and views almost always occur in pairs, many architectures combine the two functions into a single class. Recall from Chapter 4 the WinEventHandler class, which had several methods for

responding to events. The Redraw method would implement the majority of the view. (The methods to handle notification from the model and object selection for the controller must be added.) The mouse and keyboard methods would implement the controller functionality. The model is implemented based on our functional design as described in Chapter 2.

5.2.1 The Problem with Multiple Parts

In simple applications, it is tempting to combine the model, view, and controller into a single class or into global variables. Such an approach will not scale up to large applications. The model classes must be separated out for two reasons. The first is that there may be multiple models that a user is working with. In our example, the user may have an old version of the circuit on the screen and may be using it as a guide to design a new version in a separate window. This scenario would require multiple models and multiple views. The implementations would be the same but different information is being manipulated in each case.

A second problem, which is frequently ignored by those building simple applications, is the fact that a model may have more than one view. In our example, the model has at least two views, the circuit view and the parts list view. Each view is very different but each must be updated when a chip is added to the circuit. There may also be multiple, similar views of the same model. Our example application does not support scrolling of the circuit view, but let us suppose that it did. Let us also suppose that the circuit was very large and the user had need to work in two separate areas of the circuit at once. An additional circuit view of the same circuit could be created at run time. Each view could be scrolled to a different part of the circuit. In such an application, there can be any number of views of the same model, depending on what the user is trying to do. Each of these views must be kept consistent with the model and the user must be able to interact with the model through the controllers of each of those views. The support for multiple views is the primary reason for the separation between the model and the view-controller.

There are also software maintenance reasons for the separation. Suppose, for example, that our users look at our first implementation and decide that it is important to have a wiring list view that shows all of the wires and that names their connections. We could implement the new view and its controller and add it to the list of views that need to be notified whenever the model changes. The existing views would not need to be changed and the model would be unaffected. With the addition of a new view, new model information may be needed; however, the old views would still respond in the same way.

Suppose that our graphics designers and marketing people decide that chips should be drawn with a 3D look rather than a flat schematic look. Only the

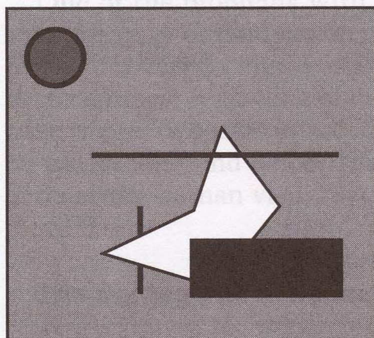

Figure 5-3 Shapes to be manipulated

view would need to be changed to draw the chips in a different way. The view would also need to be changed to select chips and contact pins in a different way, because the positions of the pins relative to the chips would be different. That is why selection tasks are handled by the view as a service to the controller. That is also why we think of the controller as conceptually different. The pattern of behavior in response to user events (controller issues) is independent of visual geometry (view issues).

5.2.2 Changing the Display

In most of our applications, any interactive work by the user will cause the model to change. In response to this change in the model, the views will need to update what is drawn on the screen. Before we go through the event flow between models, views, and controllers, we first need to work through the relationship between a view and the windowing system in handling updates to the display.

Let us consider the problem in Figure 5-3. In this example, our model consists of a list of the shapes that we want to draw, along with their colors and geometric information. We want to interact with this model by moving shapes around. The problem that our view code must solve is to change the display in such a way that the polygon stays in front of the background and vertical line as well as behind the horizontal line and the black rectangle.

One simple-minded way to solve this problem is to draw the shape being moved using the color of the background. Drawing in the background color will erase the shape in its old position. We can then draw the shape in the new position. This will work just fine in the case where we move the circle as shown in Figure 5-4.

It will not work, however, if we want to move the white polygon. The results of such an approach are shown in Figure 5-5. In this case, the drawing

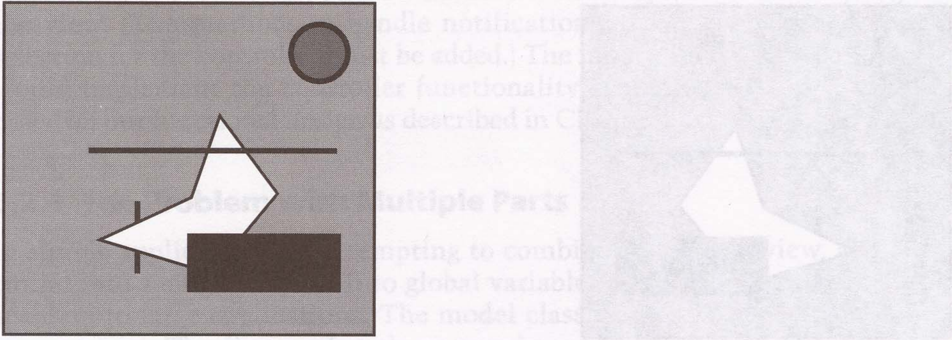

Figure 5-4 Erasing and redrawing the circle

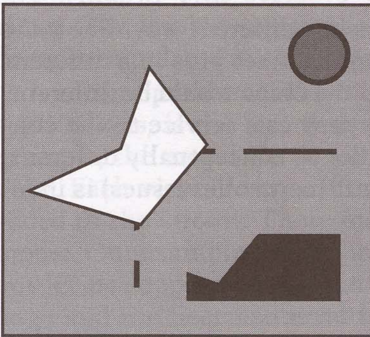

Figure 5-5 Erasing and redrawing the polygon

of the old polygon using background color has wiped out parts of the lines and the black rectangle. In addition, the drawing of the new polygon is now in front of the horizontal line, which is not correct.

An alternative to this strategy is to move the polygon in the model to its new position and then to redraw the entire picture from the model in the following order: 1) background, 2) circle, 3) vertical line, 4) polygon, 5) horizontal line, and 6) black rectangle.

By drawing the shapes in this prescribed order, the objects that are in front are drawn last and will thus overlay any objects that are behind. Such a back-to-front drawing technique will guarantee the correct drawing. In fact, in most drawing systems, the model will maintain the list of shapes in back-to-front order so as to simplify this technique. Menu actions such as "Move to Back" or "Move to Front" found in most drawing packages simply involve changing the position of the selected shapes in the list of shapes and then redrawing.

One of the problems with this complete redraw strategy is that it is too slow for large or complex drawings. The changes required to the display are frequently very localized and redrawing the entire display is a waste. In addition, complete redrawing of the entire display can cause annoying flashes each time the redraw is done because the frontmost items are momentarily erased by the background before being redrawn. This is very bothersome to users because the human visual system is tuned to pay attention when it perceives motion.

The Damage/Redraw Technique

The common technique for handling the problem of correctly updating the display uses a pair of operations that we will call Damage and Redraw. All modern windowing systems support a variant of the damage/redraw technique. Using this technique, a view can inform the windowing system when a region of a window needs to be updated. The windowing system will then batch these updates, clip them to the portions of the window that are actually visible, and then invoke the Redraw method for the window. The Redraw method is passed the window region that needs to be redrawn. This Redraw method was discussed in Chapter 4 as part of the WinEventHandler class.

In order to accommodate this technique, we need to add the Damage method to our abstract Canvas class:

```
void Canvas::Damage(UpdateRegion)
```

When a view invokes Damage on a canvas, the windowing system will save the UpdateRegion for later. One of the reasons for saving the damaged regions is that many times a model change will cause a variety of changes to the screen, which may or may not overlap. For this reason, a windowing system will save them all until the event handler requests the next input event. At that time, the Redraw methods for all windows that have changes can be invoked.

Using this technique, we can reconsider our problem of moving the polygon. When the polygon is moved, we first damage the region where the polygon used to be, so that the area can be correctly redrawn without the polygon. We then change the polygon's position in the model and then damage the region around the polygon's new position so that the new area will be redrawn.

Before any input events are handled, the windowing system will invoke the Redraw method for this window, which will redraw the damaged regions in back-to-front order. Figure 5-6 shows the damaged regions as dotted rectangles.

In our simple set of shapes, the Redraw method may just redraw the entire model in front-to-back order because the numbers are so small. The windowing system will clip to the damaged region. This clipping prevents the circle

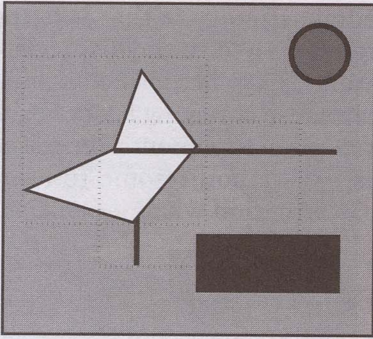

Figure 5-6 Damage/Redraw method showing damaged regions

and most of the background from actually being drawn on the screen. If, however, there were a large number of shapes in the model, the Redraw method could check groups of shapes or separate areas of the drawing against the damaged area to avoid even considering parts of the model that would not affect the damaged area. This would be much more efficient with large models.

5.2.3 General Event Flow

Having discussed the relationship between a view and the windowing system, we need to consider the entire process of handling input events, including changing the model and updating the screen. To get this overall view of the MVC architecture, we will work through a couple of interactive tasks in our example application.

Creating a New Chip

Let us first consider the creation of a new chip. We will assume that the user has already selected the chip icon on the screen and that the circuit controller has a field that remembers that the chip icon is selected. (Note that the view and the controller must share this field so that the view can highlight the currently selected icon.) The process involves the following steps:

1. To create the new chip, the user will place the mouse over the tentative position where the new chip is to go and then press the mouse button.
2. When the mouse button is pressed, the windowing system will identify which window should receive the event and locate the WinEventHandler that should receive the event. The WinEventHandler that implements our circuit view and controller will have its MouseDown method invoked. This is part of the controller.

3. The controller determines that it is in chip mode (based on the selected icon) and inquires of the view as to whether the mouse is over an existing chip. If the mouse is not over an existing chip, the controller decides that a new chip is to be created. It requests the view to start echoing a rubber band rectangle where the new chip will be placed and saves the fact that it is creating a new chip. The MouseDown method then returns.
4. The user can then adjust where the chip will be placed by moving the mouse while holding down the mouse button. Each time the mouse moves, the windowing system will invoke the controller's MouseMove method. The controller will then have the view move the echoing rectangle to the new position.
5. When the user finally decides that the chip is in the right position, the mouse button is released and the windowing system will invoke the MouseUp method on the view-controller. The controller will have the view remove the echoing rectangle from the screen, take itself out of chip-positioning mode, and invoke the AddChip method on the circuit model, passing in the new location.
6. When the model has its AddChip method invoked, it will add the new chip to its array of chips and will then go to the list of views that have been registered with this model. For each of these views, the model will invoke the appropriate methods to notify them that a new chip has been added.
7. When the part list view receives notification that there is a new chip, it will inform the windowing system that the space at the bottom of the list is damaged and needs to be updated. *Note that the part list view does not draw the new chip into the window at this point.*
8. When the circuit view receives notification of the new chip, it will also inform the windowing system that the region where the new chip is to go is damaged. *Note that even though the circuit view's controller initiated the request to create a new chip, the view still waits for notification from the model.* Suppose, for example, that the model was enforcing some design constraints that would not allow chips to overlap each other. The original position from the user might violate those constraints. The model may then move the chip slightly to accommodate the constraints. In such a case, the view must accurately reflect what is in the model, even if it is different from what the view's own controller specified. Also note that the circuit view must respond to notifications of new chips, no matter where such changes originate. By placing code to damage the window inside of the controller, such code would be duplicated.

9. When all views have been notified and have performed their damage processing, the model returns from its AddChip method to the controller, which then returns from its MouseDown method, leaving the windowing system in control again. The windowing system determines that there are damage requests pending and will respond to them. The first damage request is from the part list view. The windowing system determines, however, that this portion of the part list window is completely obscured by some other window. In this case, the damage request is discarded because the damaged region is not visible. This is why the part list view or any other view only damages the changed area in response to notification of a model change, rather than drawing the changed information immediately. The other damage request found by the windowing system is for the circuit view window. This area is not obscured, so the windowing system invokes the circuit view's Redraw method with the damaged area.
10. When the circuit view receives its Redraw message, it will look through all of the chips in the model and draw any chip that overlaps the damaged area. It will then look through all of the wires and draw any wire that appears in the damaged area. Because the windowing system sets the clip region to the damaged area, the circuit view may for simplicity draw all chips and all wires, leaving the clipping logic to discard anything outside of the damage area. Either strategy will work, although in very large circuits, the "redraw everything" approach may be too slow for interactive use. Let us suppose that our new chip has been placed over existing wires. In our application, we always want wires on top so that we can see them. If the circuit view had simply drawn the new chip when it received the notification from the model, the chip would have appeared over the top of the wires, which is not desired. By having the notification only report a damaged region, and then letting Redraw handle the rest, a correct presentation will always occur.

Moving a Chip

To further illustrate the issues of how the MVC and damage/redraw mechanisms work, let's look at a second example. In this case, we want to move the XOR chip to a new location. Remember that in our application, when we move a chip, the wires stay connected. We will start from Figure 5-7 where the chip icon has been selected. The process involves the following steps:

1. When the mouse button goes down over the XOR chip, the windowing system invokes the controller's MouseDown event.
2. The controller requests the view to select a chip and the view returns the index of the XOR chip as the one selected. The controller then notifies the model of the selection by calling the model's SelectChip method.

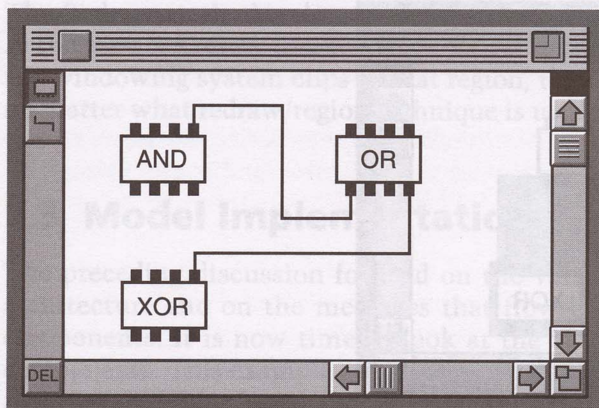

Figure 5-7 Dragging a chip

3. The model's SelectChip method notifies all views registered with that model that the XOR chip has been selected. Each view then damages its presentation of the XOR chip. In the layout view, the rectangular region around the chip is damaged; in the part list view, the chip's name region is damaged.
4. The controller then stores the fact that it is waiting to drag the chip to a new location and returns to the windowing system.
5. The windowing system locates the entries for the damaged entries and invokes Redraw methods on the appropriate views. These Redraw methods will draw the presentation of the XOR chip to show that it has been selected.
6. The windowing system then waits for more input events. Since we are dragging the chip, the next input event will be a movement of the mouse. When each mouse movement is received by the windowing system, the system calls the MouseMove method on the circuit layout view. This method must echo the new location of the chip on the screen. The normal notify/damage/redraw cycle is frequently too slow for this type of echo. Later in this chapter we will discuss faster echoing mechanisms that the controller can use without involving the model or the view.
7. When the mouse button is released, the windowing system will send a MouseUp message to the controller. The controller remembers that it is dragging a chip to a new location and invokes the model's MoveChip method.
8. The model will notify each view that the XOR chip has moved to a new location. The part list view will ignore this notice because its display does not involve the chip location.

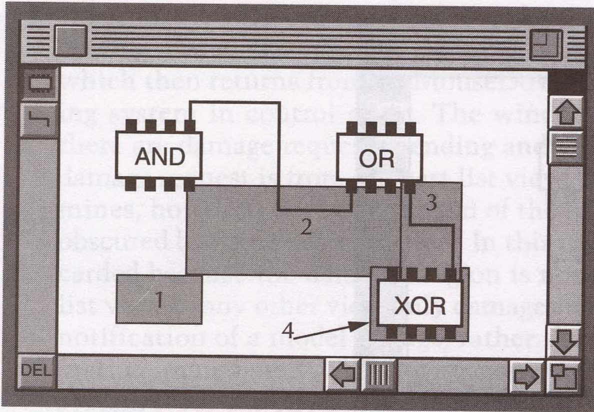

Figure 5-8 Damage regions to move a chip

9. The circuit layout view, however, has some work to do. The circuit layout view must not only move the chip but also the wires connected to it as well. When moving an object, we must damage both the old location and the new location. Because wires are moving, the areas around the wires must also be damaged. Figure 5-8 shows the chip and wires in their new locations. The gray rectangles show all of the regions that must be damaged to correctly redraw the view.
10. After the model has notified all of its views and has changed itself, it returns to the controller. The controller takes itself out of dragging mode and returns to the windowing system.
11. The windowing system locates the damaged entries and invokes the correct Redraw methods. When region 1 (see Figure 5-8) is redrawn, the view checks the model and detects that there is nothing in that region. The area is drawn in background color. When region 2 is redrawn, the view detects that a portion of the wire from the OR chip to the AND chip must be redrawn. The rest is background. These two redraws will cause the XOR chip and its wires to disappear from their old positions. When region 3 is redrawn, the wire in the new position is drawn; region 4 draws the chip in its new position.

In some windowing systems, the Redraw method would be called four times, once for each rectangle. In other systems, the process might be batched together into one large rectangle that encloses all four damaged regions. The Redraw method is then invoked only once with the large rectangle. In other windowing systems, the four rectangles would be assembled into a single complex region that exactly bounds the area specified by the four rectangles.

The Redraw method is then invoked once with this complex region. As long as the view's Redraw method can correctly redraw any region and as long as the windowing system clips to that region, the screen updates will be correct no matter what redraw/region technique is used by the windowing system.

5.3 Model Implementation

The preceding discussion focused on the various components of the MVC architecture and on the messages that flow back and forth between those components. It is now time to look at the actual implementation of those components. This example approach is not the only implementation strategy. At the end of this chapter, we will discuss variations on the theme in various commercial tool kits.

We will start our implementation discussion with the model. There are two aspects that need to be considered. The first is the interface that the model will present to the views and controllers. The second is the mechanism for the model to notify all views of changes to the model.

5.3.1 Circuit Class

As described earlier, the heart of our model is the Circuit class which, in conjunction with the Chip and Wire classes, represents everything that our application needs to know about circuits.

The methods are

```
void Circuit::AddChip(CenterPoint)
void Circuit::AddWire(Chip1, Connector1, Chip2, Connector2)
void Circuit::SelectChip(ChipNum)
void Circuit::MoveChip(ChipNum, NewCenterPoint)
void Circuit::ChangeChipName(ChipNum, NewName)
void Circuit::DeleteChip(ChipNum)
void Circuit::SelectWire(WireNum)
void Circuit::DeleteWire(WireNum)
```

and the fields are

```
Chips
Wires
SelectedChip
SelectedWire
```

These are drawn directly from the functional model that we designed earlier. The key to this model class is the methods. These are the methods that views, controllers, and other software in the application will call to make changes to the model. These methods will perform the normal data structure updates to the model, but in addition, they must also notify the views of any changes. It is this change notification that needs particular discussion here.

5.3.2 CircuitView Class

The Circuit class needs a general mechanism for notifying all of its views that some aspect of the circuit has changed. We do not want to code into our model the information about every type of view of the model. Such hard coding would seriously hamper our ability to add new views or to modify existing views. To provide this generality, we define the abstract class CircuitView. This class will define the Circuit class's interface to any of its views. The CircuitView class and its subclasses will combine view and controller functions in a single class.

The CircuitView class must be a subclass of WinEventHandler (as defined in Chapter 4), so that it can receive input and redraw events from the windowing system. Each of our views will be subclasses of CircuitView and thus will inherit both the windowing system interface methods and the view interface for the Circuit class.

Model Registration

The CircuitView class will have one attribute, which is myCircuit and is of type Circuit. This attribute provides each view with a pointer to the model that it represents. When a new CircuitView is created, it must be given a pointer to its model and a pointer to a Canvas that it is to draw upon.

Notification Methods

The methods that we add to our CircuitView class are the ones that the Circuit class will need to inform the view of any changes, as follows:

```
void CircuitView::ChangeChip(ChipNum)
```
The specified chip has been changed.

```
void CircuitView::ChangeWire(WireNum)
```
The specified wire has been changed.

```
void CircuitView::ChipMoved(ChipNum, NewLocation)
```
The chip is being moved to a new location.

These three methods will provide all of the view notification that we need. When a view receives the ChangeChip message, it will damage any regions that display information about that chip. Subsequent redraws will draw those

regions with the new model information about that chip. For the same reason, the ChangeWire method will damage the region where the wire is drawn.

The ChipMoved method is special because it must damage the chip and its wires at their old locations as well as their new locations. The old locations can be determined by looking at the model. The new location can be determined from NewLocation. For this technique to work, ChipMoved notifications must be sent before the model is changed so that the old information is still available.

In the abstract class CircuitView, these view notification methods do nothing at all. Their actual behavior depends on the specific view. The damage work required for ChangeChip in the circuit layout view is very different from ChangeChip on the parts list view. When we create subclasses of CircuitView, we will provide implementations for these methods.

5.3.3 View Notification in the Circuit Class

A major function of the Circuit model is to notify the views of any changes. We need to consider this as we look at each of the Circuit methods. Before discussing those methods, however, the Circuit class needs a mechanism for keeping track of which views are attached to each model object. We need to add a list of CircuitView objects to the Circuit class, along with the following two methods:

```
void Circuit::AddView(View)
void Circuit::RemoveView(View)
```

These two methods will manage the list of views. In addition, we need a mechanism for notifying each view of the changes. To do this, we add the following methods:

```
void Circuit::ChangeChip(ChipNum)
void Circuit::ChangeWire(WireNum)
void Circuit::ChipMoved(ChipNum, NewLocation)
```

These methods are the same notification methods found on the CircuitView class. The model code can invoke these notification methods on the Circuit itself without worrying about the views that need to be updated. Each of these Circuit methods will loop through the list of views in the circuit's view list and invoke the corresponding method on each view. For example, the Circuit::ChangeChip method would be

```
void Circuit::ChangeChip(ChipNum)
  { for each V in the circuit's view list
     { V.ChangeChip(ChipNum) }
  }
```

Now that we have our view notification mechanisms designed and in place, we can discuss the actual methods that implement the circuit's functional model.

The AddChip method will first add a new chip to the end of the list at index *I*. The method will then call ChangeChip(*I*), which will notify all views that this new chip needs to be redrawn. The AddWire method works in the same way except that it calls ChangeWire(*I*) after the wire is in place. In both cases, the view can look at the model attributes to find the necessary information about the new addition.

The SelectChip and SelectWire methods change the selection from one object to another. The notification for SelectChip must first be ChangeChip(OldSelection) and then ChangeChip(NewSelection). The operation is similar for SelectWire. This can be optimized slightly by checking to see if OldSelection and NewSelection are identical, in which case no notification is required.

The MoveChip method on Circuit must use the ChipMoved notification. The reason the ChangeChip notification is not sufficient is that moving a chip requires more than just damaging the chip's bounding rectangle. The new and old wire positions must also be calculated. The CircuitView::ChipMoved method will take care of all of this for us.

When deleting objects, as in DeleteChip and DeleteWire, we must call the ChangeChip or ChangeWire notification before the deletion. Remember that views only perform damage operations, leaving all drawing to the Redraw method. The ChangeChip method will look in the model for the chip to be deleted and will damage its region. After ChangeChip returns, the chip is deleted. Later, the windowing system will process a Redraw on that region; at that time, the chip will no longer be in the model and the region will be drawn blank. The DeleteChip method is further complicated by the fact that all wires connected to the chip must be deleted. The DeleteChip method can use DeleteWire to do this, and the DeleteWire method will notify the views of the deleted wires. To accomplish all of this takes the following steps:

1. DeleteChip will call ChangeChip (for view notification).
2. DeleteChip then calls DeleteWire for all wires connected to the chip.
3. Each invocation of DeleteWire will call ChangeWire (for view notification) and will remove the wire from the model.
4. After all connecting wires are deleted, DeleteChip will remove the chip from the model.

5.3.4 Overview of the Circuit Class

The Circuit class implements the model for our application. It provides methods for registering any number of views with a circuit model and it provides

methods that will send the notification messages to all views in the list. These notification messages are defined on the CircuitView class that defines the model/view interface. In addition, Circuit has methods that implement the model functionality and that notify the views of any changes.

Note that in a more complex application, there may be multiple model classes. Each of these models may or may not share the same view notification class. There may be several classes like CircuitView that each define a view notification interface for a particular type of model.

5.4 View/Controller Implementation

Up to this point, we have only defined the model and its interface to its views. With our model implementation in place, we can work on the views and controllers. The architecture used in this book will combine the concepts of view and controller into a single class. In our example application, all views are a subclass of CircuitView and as such can provide the uniform notification interface required by the Circuit class. Any view that is a subclass of CircuitView can be passed to Circuit::AddView and thus can be attached to a circuit model. Each view class also inherits from CircuitView a pointer to the circuit object that implements a view's model.

There are four issues that must be considered when implementing views. The first is the actual drawing of the view in response to a Redraw message. The second is responding to change notification messages from the model. The third is the handling of selection, which is the translation of a mouse location into some record of what should be selected. These first three issues make up what is considered to be a view in the MVC architecture. They are all concerned with the geometry of the presentation. The fourth issue is the controller implementation that translates the input events into calls on the model. The controller translates user input into the methods of the functional model.

5.4.1 PartListView Class

The PartListView class implements the view of a list of the parts in our circuit layout. It is shown in Figure 5-9. This view is simplified by its regular geometry. The chips are presented in their part number order, which is the same order as they have in the model. Each chip's presentation is exactly the same height. In fact, we can exploit the regularity by defining the following new method:

```
Rectangle PartListView::ChipArea(ChipNum)
```

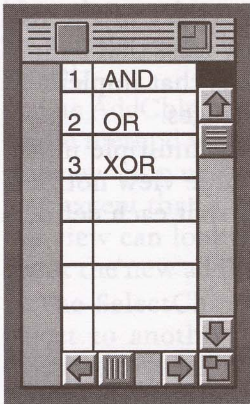

Figure 5-9 Part list view

This method will take a chip number and will return the rectangle that bounds the area of that chip. Because PartListView is a subclass of WinEventHandler, it has a GetBounds method that will return the bounding rectangle for this window. Let us assume that we know the height in pixels (CH) of each chip in the view. With this information, we can compute the new rectangle to be

```
B = GetBounds()
Top = B.top + (ChipNum-1) * CH
Left = B.left
Bottom = B.top + ChipNum * CH
Right = B.right
```

In most view classes, there is some essential geometry, like ChipArea, that defines the presentation geometry. This essential geometry will be used throughout the class. It is very helpful to identify these aspects of a view and to implement them as separate methods for use by the rest of the methods in the class. We will see this again in our LayoutView class.

Selection

We will address selection first because such methods can provide geometry logic that we can use in other ways. There are two selection problems that must be addressed by the view. The first is the selection of the correct chip and the second is the selection of a character in the chip's name. The following method can implement our chip selection:

```
long PartListView::WhichChip(MouseLoc)
    { return MouseLoc.Y - GetBounds().Top) / CH + 1;
    }
```

This method translates mouse positions into an index of which chip is selected.

Having selected the correct chip, we need to select the index of the character in the name that lies under the mouse position. We can implement this PartView::WhichChar method using the techniques for text selection that were defined in Chapter 3.

Presentation Drawing

The drawing of a view's presentation is defined by its Redraw method. A simple version of the Redraw method might be as follows:

```
for each chip C in the model
  { R = ChipArea(C);
     draw info about chip C in R
     draw horizontal line across the bottom of R
  }
Draw bounding rectangle for the window.
Draw the vertical line.
```

Drawing each chip is a matter of drawing the chip number and name from the model using the Canvas.

This simple version of Redraw will draw the entire window in response to every required update. It relies upon the windowing system to clip all of the output to the actual region being redrawn. Such a model will work correctly and, in the case of small numbers of parts in the list, is usually fast enough for interactive purposes. Remember, however, that the Redraw method receives as an argument the actual region that needs to be updated. Because of the simple geometry of our PartListView, we can use this region information to create a more efficient redraw.

Let us suppose that the redraw region (RR) has a method for computing the top and bottom of the region. Remember that clipping regions are not necessarily rectangles. Given such methods, we can provide the following Redraw method:

```
TC = WhichChip(RR.top)
BC = WhichChip(RR.bottom)

for each chip C from TC to BC
  { R = ChipArea(C);
     draw info for chip C in R
     draw horizontal line across the bottom of R
  }
Draw bounding rectangle for the window.
Draw the vertical line.
```

This revised version of the Redraw method will only draw those chips that may actually be involved in the region to be updated. Note that we still draw the complete border and vertical line because the clipping of such simple objects is more efficiently done by the windowing system. Our optimization exploits the additional knowledge we have of the geometry of our presentation.

Change Notification

The change notification methods inherited from CircuitView include ChangeChip, ChangeWire, and MoveChip. Our PartListView is only concerned with the ChangeChip method because it does not display wires and it does not display anything about the geometry of chips. The PartListView class will override the ChangeChip method to provide its own implementation. The other two methods will be inherited and as such will do nothing.

The implementation of ChangeChip is quite straightforward, using our ChipArea method:

```
void PartListView::ChangeChip(ChipNum)
  { myCanvas.Damage(ChipArea(ChipNum)) }
```

Controller

The controller is that aspect of the PartListView class that handles input events. Its functionality is spread across the methods inherited from WinEventHandler that receive input events. In our simple event model, these are

MouseDown
MouseMove
MouseUp
KeyPress
MouseEnter
MouseExit

For this discussion, we will ignore MouseEnter and MouseExit.

There are only two things that a user can do in the part list view. The first is to select a new chip and the second is to change the name of a chip.

A very simple way to implement chip selection uses only the MouseUp method:

```
void PartListView::MouseUp(location, modifiers)
  { myCircuit.SelectChip(WhichChip(location)); }
```

Whenever the mouse is clicked in the part list window, the chip under the mouse position will be the newly selected chip. One of the problems our users will have, however, is that they will not always know which chip in the list

relates to which chip on the layout. Being able to drag the mouse over the part list while watching the selected chip in the layout can be very helpful. We can do this by implementing MouseDown and MouseMove, as follows:

```
void PartListView::MouseDown(button, location, modifiers)
  { myCircuit.SelectChip(WhichChip(location)); }
void PartListView::MouseMove(location, modifiers)
  { if (modifiers show button is down)
     { myCircuit.SelectChip(WhichChip(location));
     }
  }
```

This new implementation will change the selection only while the mouse is down. Note that in MouseMove we need to check to see if the button is currently pressed, because the MouseMove method is called whether the button is down or not, and we don't want to change the selected chip every time the mouse casually crosses the part list.

We still have not handled the changing of the chip names, however. To change chip names, we must handle the selection of the insert point in the selected chip's name and we must handle keyboard events. Before we can handle the selection point, our PartListView class needs a new attribute, CharInsertPoint, that will remember which character was selected.

The selection point can be handled by modifying MouseUp to identify and save the selected character:

```
void PartListView::MouseUp(location, modifiers)
  { myCircuit.SelectChip(WhichChip(location));
    CharInsertPoint = WhichChar(location);
  }
```

The actual editing of the names is performed in the KeyPress method. Each time the user presses a key, the windowing system invokes the view's KeyPress method. There are several cases that the KeyPress method must handle, as follows:

1. There is no selected chip.
   Return and do nothing.
2. The current character index is 0 and the input character is a backspace.
   Do nothing; you can't delete at the beginning of a string.
3. The current character index is greater than 0 and the input character is a backspace.
   Delete the indexed character from the string, invoke the model's ChangeChipName method, and decrement the current character index.

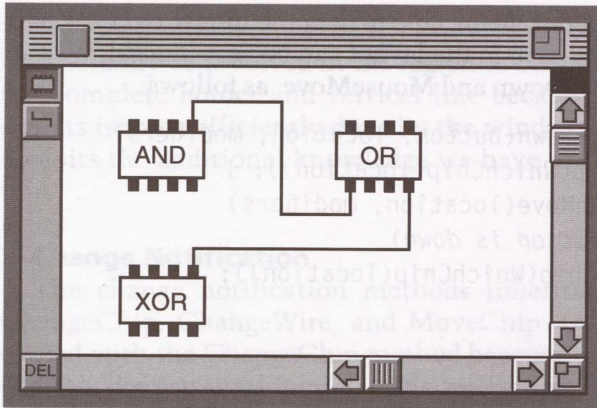

Figure 5-10 Layout view

4. The input character is not a backspace.
   Add the input character to the string, invoke the model's ChangeChipName method, and increment the current character index.

The KeyPress method illustrates the structure of most of the controller methods. A set of cases must be defined to decide what actions to take. The cases are determined by the current input (input character), the state of the model (which chip is selected), and information stored by the controller itself. In Chapter 8 on input syntax, we will discuss more precise methods for capturing these cases and translating them into code.

5.4.2 LayoutView Class

The LayoutView class is more complicated because the geometry does not have the nice array structure found in the part list. In fact, as shown in Figure 5-10, there are actually two levels of geometry in the layout view—sorting out the icons from the layout area and the actual layout of the chips and wires.

Because the layout view is more complex, we need to decompose its structure into simpler parts. At the first level of decomposition, we can think of the layout view as a drawing and a set of icons. Each icon is differentiated by its picture and by what it does when selected. Within the drawing there are chips and wires. Looking at this decomposition, we can devise several new classes from which we can build the layout view. For this design, we will create the Icon, Drawing, ChipV, and WireV classes.

Note that with most modern user interface tool kits we would not implement our own chip, wire, and delete buttons. Instead, we normally use the button widgets provided by the tool kit. We are going to delay discussion of widget design and use until Chapter 6 so as to first understand the mechanisms that underlie such tool kits.

Rather than design these classes at this point, we will work through the LayoutView class implementation to understand what we will need each of these helper classes to do for us. Our LayoutView class will need three Icon attributes and one Drawing attribute:

```
Icon        ChipIcon;
Icon        WireIcon;
Icon        DeleteIcon;
Drawing     CircuitDrawing;
```

Selection

The selection logic for the LayoutView actually consists of delegating the work to the various helper classes. Let us assume that each of the helper classes has a Select method such as

boolean *class*::Select(MousePoint);

This method returns true if the mouse position would select the object. Based on this method, the LayoutView class can send the Select message to each of the icons and drawing area in turn. The first object whose Select method returns true is the one selected. The Drawing class can use the same approach when it needs to select chips or wires. Objects in the drawing are selected by testing the Select method on each one of them.

Presentation Drawing

The presentation drawing is the visual representation of the model in the view. The presentation drawing for the LayoutView is defined by its Redraw method. Let us assume that each of the helper classes has the following two methods:

Rectangle *class*::Bounds();

void *class*::Redraw(DamagedRegion);

If each helper class has these methods, then the redraw for the layout view is quite simple:

```
void LayoutView::Redraw(Damaged)
  { if(ChipIcon.Bounds() intersects Damaged)
      { ChipIcon.Redraw(Damaged); }
    if(WireIcon.Bounds() intersects Damaged)
      { WireIcon.Redraw(Damaged); }
    if(DeleteIcon.Bounds() intersects Damaged)
      { DeleteIcon.Redraw(Damaged); }
    if(Drawing.Bounds() intersects Damaged)
      { Drawing.Redraw(Damaged); }
  }
```

The decomposition of the problem into the helper objects greatly simplifies the complexity of our problem. We will use this technique in numerous places.

Change Notification

Remember that the model provides the ChangeChip, ChangeWire, and MoveChip notifications. In response to a ChangeChip message, the view need only damage the bounding rectangle of the chip. Since there are classes for chips and wires that support the Bounds method, we need only invoke Bounds on the relevant chip or wire view object and then damage the returned rectangle.

In response to a MoveChip notification, the old chip-bounding rectangle must first be damaged. Because wires will also be moved, the view must run through the model and damage the bounds of every wire that is connected to the chip being moved. Next, the chip-bounding rectangle is damaged in its new location. Finally, the bounds of all connecting wires are damaged in their new locations. Damaging the chip and wires is relatively easy, given the Bounds method on the ChipV and WireV classes.

Controller

Before designing the controller, we need to consider the interactive behavior of the LayoutView class. There are four interactive tasks that the layout view supports:

- selecting chip or wire mode
- deleting a chip or wire
- creating a new chip or wire
- dragging a chip to a new location

The LayoutView class is considerably more complex in its behavior than PartListView. In PartListView, each interactive task was essentially handled by one of the input-event–handling methods. In the LayoutView, each of the

four interactive tasks is scattered over the MouseDown, MouseMove, and MouseUp methods. In addition, the behavior is also partitioned among the helper classes. In order to understand this behavior, we will first work through the interactive behavior of each task and then look at how the behavior is partitioned among the various class methods.

Selecting Modes Selecting chip mode or wire mode is a process of selecting the right icon. A very simple interactive behavior is to select an icon when a MouseUp event occurs: when MouseUp is received, we test the mouse position using ChipIcon.Select and WireIcon.Select; when either of these returns true, the LayoutView must remember which one (chip or wire) was selected. We can handle this with a new field on the LayoutView class:

```
enum{ ChipMode,WireMode } IconMode;
```

Note also that if IconMode is changed, the ChipIcon and WireIcon objects must be damaged so that they can correctly draw their highlights to show the selected mode.

Normally, such damage actions would be in response to change notification from the model. The IconMode information, however, is not a property of the model; it is a property of the view. As such, the view itself must account for any screen updates. This behavior can be understood as two cases:

1. MouseUp and ChipIcon.Selected
   IconMode = ChipMode
   Damage bounds of ChipIcon and WireIcon.
2. MouseUp and WireIcon.Selected
   IconMode = WireMode
   Damage bounds of ChipIcon and WireIcon.

Deletions Deletion can also be handled in MouseUp by adding a test of the DeleteIcon.Select method. In this case, the LayoutView will consult the SelectedChip and SelectedWire fields in the model to determine which objects should be deleted. Invoking the DeleteChip and DeleteWire methods on the model will cause the model to notify the views, which will cause areas of the display to be damaged, which will then cause the windowing system to send redraw messages to the various views.

This behavior can be represented as a third case:

3. MouseUp and DeleteIcon.Selected
   Model.DeleteChip(Model.SelectedChip)
   Model.DeleteWire(Model.SelectedWire)

Creating Chips or Wires There are several cases involved in this task. The behavior we want is to begin drawing the desired chip or wire when the

mouse button goes down. While the mouse button is being held down and the mouse moves, we want to draw the new chip or wire in the new position so that the user can see where the chip or wire will ultimately end up. When the mouse button is released, we want to notify the model to create the new chip or wire. The task is driven by the input event, the IconMode, and by whatever is being selected by the mouse location, as shown in the following seven cases:

4. MouseDown and IconMode == ChipMode and *there is no chip being selected.*
   Draw the new chip.
   Remember the new chip location.
5. MouseMove and *we are creating a chip.*
   Erase the chip in the old location.
   Draw the chip in the new location.
   Remember the new location.
6. MouseUp and *we are creating a chip.*
   Erase the chip in the old location.
   Model.AddChip(MouseLocation).
7. MouseDown and IconMode == WireMode and *there is no chip being selected and the mouse is over a chip connector.*
   Draw the new wire.
   Remember the new wire's connector.
8. MouseMove and *we are creating a wire.*
   Erase the wire in the old location.
   Draw the wire in the new location.
   Remember the new wire end point.
9. MouseUp and *we are creating a wire* and *the mouse is over a chip connector.*
   Erase the wire in the old location.
   Model.AddWire(. . .).
10. MouseUp and *we are creating a wire* and *the mouse is* **not** *over a chip connector.*
    Erase the wire in the old location.
    Forget that a wire is being created.

Dragging Chips The task of dragging a chip begins with the selection of a chip upon mouse down and the redrawing of the chip with each mouse movement until the mouse is released. The cases for this task are as follows:

11. MouseDown and *the mouse is over a chip to be selected.*
    Select the chip and remember that the controller is in dragging mode.

12. MouseMove and *the controller is dragging.*
    Erase the chip in its old location.
    Redraw the chip in its new location.
13. MouseUp and *the controller is dragging.*
    Erase the chip in its old location.
    Invoke the MoveChip method on the model.

Techniques for Dragging at Interactive Speeds The dragging and creation of chips and wires requires that objects be erased in their old location and redrawn in their new location each time the mouse is moved. As discussed earlier, the move/redraw cycle must take less than 1/5th of a second per cycle for the user to feel comfortable with continuous movement. The problem lies in getting the image redrawn and erased fast enough.

The simplistic method is to damage the old and new areas and to let the windowing system take care of the problem using redraw. Unfortunately, on many machines this is not fast enough. Even on very fast machines, some models may be too complex for fast enough redraw.

A technique that is frequently used is to not redraw the actual image of the object that is being moved but rather to use a cartoon of the object. In the case of dragging a chip, we might only drag its bounding rectangle without updating its connecting wires. This is enough of a hint to the user in placing the chip that we do not need to draw all of the details. When the model is then updated, the normal redraw cycle will correctly draw the resulting movement.

Even when using a cartoon of the object that is being moved, there is still the problem of erasing the cartoon in its old position before drawing it in its new position. On the old vector-refresh displays, this was not a problem because changing the image would change the display in one refresh cycle. On frame buffer displays, erasure as well as drawing will damage the frame buffer and mess up whatever else should be drawn underneath.

A common erase/redraw technique is the use of exclusive-OR (XOR) drawing mode. This takes advantage of the following property of the XOR operation:

for any values of D and P

$$P' = \text{XOR}(D, P)$$

$$P = \text{XOR}(D, P')$$

These properties mean that for any value D, when we draw into pixel P using an XOR, that pixel will change to P'. If we draw D again into that pixel using XOR, the value P' will change back to P. We can first draw our cartoon using D with the XOR operation, which will change all of the pixels in our cartoon to

some P' depending on what the original pixel color was. When it comes time in the next cycle to erase the cartoon, we can draw it again in XOR mode and the pixels will return to their original color.

Using XOR mode to draw a white object into frame buffers that have only a single bit per pixel will complement the values of the pixels, making white appear on black and black on white. This provides a nice contrast and works very effectively. In the case of true color displays with 24 bits per pixel, this process still works well in the case of very dark or very light colors because the complement has a high contrast to the original. In the case of neutral colors, however, the complement is a very similar neutral color and the cartoon does not have sufficient contrast to be easily visible during dragging.

The contrast problem is particularly acute when using frame buffers with color-lookup tables. The XOR operation is applied to the contents of each pixel. In a lookup-table architecture, the pixel value is not a color but an index to where the color is stored. The XOR in index space may not make any visual sense at all. Again, we may end up dragging with insufficient contrast. This is very frustrating for the user because he cannot perceive what the system is doing.

There is a technique that is very fast and that can help alleviate this problem, while still allowing the use of the XOR operation. Let us assume that the color of the object we are trying to draw is C and that the normal background color we are trying to draw upon is B. What we want is to choose some color D such that $C == \text{XOR}(D, B)$. That is, whenever D is drawn over B, the resulting color will be C, which is what we are trying to draw. The color D is easily computed as $D = \text{XOR}(C, B)$. By using a D computed in this fashion, we are always guaranteed that the resulting dragging color using XOR will contrast with B. Because dragging will not always be over the background, the contrast may not always work. For example, dragging a cartoon of a chip over existing wires may not produce a contrast over the wires, but we may not care much because the contrast over the background will yield sufficient visual information to guide the user while dragging.

An alternative dragging technique is to save the portion of the frame buffer under the cartoon before drawing. The pixels are saved in a separate off-screen buffer; when the cartoon is to be erased, the pixels are drawn back as they were before the cartoon was drawn. On most modern machines, there is sufficient speed to save and restore blocks of pixels in this fashion. This pixel-saving method can provide a quite accurate erase/redraw. There are some problems, however, in network-based graphics systems, such as X, where this technique may introduce excessive network traffic into the erase/redraw cycle and may eliminate the speed advantage.

Assembling the Cases into Input Methods The 13 cases discussed above are partitioned based on the interactive tasks that need to be performed. The approach we have taken so far is very useful in understanding how each

task is designed, but it will not guide our implementation. The implementation is structured not around interactive tasks but around the MouseDown, MouseMove, and MouseUp input-event methods. Having designed our interactive tasks based on cases, we can reorganize our cases; this reorganization leads directly to an implementation for each of the input methods. Note, however, that even with our simple application, the number of cases can grow quite large. In this example application, we have also ignored the MouseEnter and MouseExit methods that account for intervals when the mouse leaves and enters the view being controlled. To handle more complex interactions, we will need more sophisticated approaches for describing and working with input behavior. Such an approach is discussed in Chapter 8, Input Syntax. The reorganized list of cases is as follows:

MouseDown

4. MouseDown and IconMode == ChipMode and *there is no chip being selected.*
   Draw the new chip.
   Remember the new chip location.
7. MouseDown and IconMode == WireMode and *there is no chip being selected* and *the mouse is over a chip connector.*
   Draw the new wire.
   Remember the new wire's connector.
11. MouseDown and *the mouse is over a chip to be selected.*
   Select the chip and remember that the controller is in dragging mode.

MouseMove

5. MouseMove and *we are creating a chip.*
   Erase the chip in the old location.
   Draw the chip in the new location.
   Remember the new location.
8. MouseMove and *we are creating a wire.*
   Erase the wire in the old location.
   Draw the wire in the new location.
   Remember the new wire end point.
12. MouseMove and *the controller is dragging.*
   Erase the chip in its old location.
   Redraw the chip in its new location.

MouseUp

1. MouseUp and ChipIcon.Selected
   IconMode = ChipMode
   Damage bounds of ChipIcon and WireIcon.

2. MouseUp and WireIcon.Selected
   IconMode = WireMode
   Damage bounds of ChipIcon and WireIcon.
3. MouseUp and DeleteIcon.Selected
   Model.DeleteChip(Model.SelectedChip)
   Model.DeleteWire(Model.SelectedWire)
6. MouseUp and *we are creating a chip.*
   Erase the chip in the old location.
   Model.AddChip(MouseLocation) . . .
9. MouseUp and *we are creating a wire* and *the mouse is over a chip connector.*
   Erase the wire in the old location.
   Model.AddWire(. . .).
10. MouseUp and *we are creating a wire* and *the mouse is not over a chip connector.*
   Erase the wire in the old location.
   Forget that a wire is being created.
13. MouseUp and *the controller is dragging.*
   Erase the chip in its old location.
   Invoke the MoveChip method on the model.

Helper Classes

Our discussion of the LayoutView class has been simplified because we have assumed the existence of the Icon, Drawing, ChipV, and WireV classes. These classes should provide a Select method that will return a true or false depending on whether the mouse position selects that object. The Select method on Drawing should return a structure indicating whether a point, a chip, or a wire has been selected. The Drawing::Select method is implemented by calls to Select on ChipV and WireV. The Select method on ChipV must return either no selection, a chip selection, or a connector of the chip that was selected.

Each of the helper classes should support a Redraw method. The Drawing::Redraw method should simply call the Redraw method on ChipV and WireV objects.

The ChipV and WireV classes pose a special problem. If we create one ChipV object for every chip in the model and one WireV for every wire in the model, each instance of a LayoutView will essentially duplicate the entire model. This not only leads to excessive memory requirements but also requires duplication of the code to manage the model's data structure. One technique to alleviate this is for the Drawing class to maintain only one

ChipV and one WireV object. Before any methods are invoked on such an object, the Drawing class first changes its chip or wire index to the correct chip or wire that the Drawing class needs help with. This simple assignment is cheap and prevents duplication of all of the chips and wires. In the case of our application, such techniques work quite well. With more complex models, we will need to be more sophisticated.

5.5 Review of Important Concepts

The overall architecture of an application is based on the Model-View-Controller (MVC) architecture designed for Smalltalk. At the heart of the architecture is the model, which is composed of those classes that implement the functional model designed for the application. The architecture used in this chapter has modified the MVC architecture by combining the view and controller functions into a single view class.

There are three basic steps in building an interactive application. The first is to design the object classes that will make up the model. The second is to design an abstract class for all views of that model. The third is to design each of the view classes.

5.5.1 Functional Model

In designing the model classes, it is very important that any changes made to the information in the model be performed by means of a method. There should be no direct modification of the model from outside of those classes, because model modification must not only change the model but must also notify all views of the changes that have occurred.

5.5.2 View Notification

The abstract view class—such as our CircuitView class—will have methods for every notification that is generated by the model. The model classes will maintain a list of CircuitView objects that are to be notified whenever the model changes. It is easiest to provide the model class with the same methods as the view class. The model's methods will invoke the view methods on every view object in the model's list of views. The purpose of this architecture is to correctly support multiple views of differing kinds to display and to interact with a given model. The model classes never interact in any way with the user or the screen. All interaction is performed by notifying the views of any changes to the model.

5.5.3 View Implementation

Each view of a particular model is implemented as a subclass of that model's view class. The view implementation consists of methods that implement essential geometry, selection methods, change notification, redraw, and the input controller methods.

Essential Geometry

In any view, there is some essential geometry that can be abstracted into a set of methods that are used by the rest of the class. It is best to first define and implement this geometry. In the PartListView, this was the location of each chip's display in the list. In the LayoutView, it was the partitioning of icons and the geometry of wires and chips. In many cases, this geometry can be broken down into classes to handle component objects that make up the view.

Selection

The selection methods provide the controller methods with essential information about the view presentation. These methods are frequently combined with the essential geometry.

Change Notification

The change notification methods are inherited from the model's view class. These methods only damage those areas of the display that need to be redrawn. Change notification methods do not do any drawing on the canvas.

Redraw

All drawing is directly performed—or indirectly performed through other methods—by the view's Redraw method. By localizing all drawing in this method, we can ensure that all drawing is done in a uniform fashion and that all display updates are consistent with the overlapping of displayed objects and with the windowing system's management of other windows.

Controller

The controller methods are inherited from WinEventHandler and manage all input events. When designing the controller, we first work out the details of each interactive task as a set of cases with their appropriate actions. We then reorder those cases by input event rather than by interactive task. These reordered cases will give us the implementation of the input methods.

The controller methods do no drawing. They use the selection methods to determine how the mouse or other inputs relate to the display and then they invoke the model's methods to make modifications to the model. The model then notifies all views, not just the one that originated the modification, of

the changes that have been made. Each view's change notification methods will damage the appropriate areas of the display. The windowing system will batch up all of the damaged areas and invoke the Redraw method on any view that needs to be updated. The Redraw method will update the display based on the current contents of the model.

5.6 An Alternative Implementation

The implementation approach used in our example exploited the use of abstract classes and the message/method–binding mechanisms of an object-oriented language to do all of the event dispatching and change notification. This approach makes it quite easy to handle the variety of messages and notifications that must pass between the objects that make up the interactive architecture.

There are some drawbacks to this architecture, however. The first is that every model must have its own abstract view class, and every view must inherit both the method interface necessary to receive input from the windowing system and the method interface needed to receive change notification. This is further complicated in more sophisticated applications when views contain other views and must also communicate with them.

In some systems, the views have only one Event method. The Event method has one parameter that is a pointer to an event record, and the event record always has as its very first field an integer event type. The remainder of the record is formatted according to the type of the event. This Event method is used to handle all input events, system notification events, model change notifications, and any other communications.

When an object receives an event, its Event method must look at the type of event to sort out what should be done. Any event that it does not understand is forwarded to its superclass's Event method. This architecture eliminates the need for the abstract view class because any event-handling class can receive the Event message. Each model can then formulate its own notification events and forward them through the Event method on any view in the model's event list. This approach allows for a single Model class that contains all of the methods and fields for registering views. The Model class can also provide a ViewNotify method that will forward an event on to every view in the list of views registered on the model.

This alternative implementation is much more flexible than the strictly object-oriented model described in this chapter. In exchange for the flexibility, however, most of the type checking and event management must be done by the programmer rather than the compiler. It is a trade-off that must be carefully considered when building new applications.

5.7 Visual C++

In order to get a more commercial slant on the concepts discussed in this chapter, we can look at the View/Document architecture of Visual C++. All drawing is handled by means of the CDC (device context) class, which provides a generic superclass for all windows, printers, and other areas that a program might want to draw into. This is identical to our concept of a Canvas. The CView class is the superclass for all views and roughly corresponds to our WinEventHandler. Visual C++ does not use the abstract view classes, such as CircuitView, in its change notification model. The CDocument class defines the superclass for all models. The terminology for Visual C++ is as follows:

- WinEventHandler is a View.
- Canvas is a CDC (device context).
- Model is a Document.

5.7.1 CView

The CView class works pretty much the way we have discussed for the functioning of views. There is an OnDraw method that performs the Redraw function. The OnDraw method, however, receives a device context as its parameter rather than a damaged region. In order to determine what has been damaged, the programmer can question the device context. This allows a single view object to display on both a printer and a window because the view is not explicitly bound to a drawing surface until the OnDraw method is called.

The CView class provides an Invalidate method, which performs the same function as Damage. This directly mirrors our architecture for updating the screen.

The Visual C++ approach to change notification from the model, or document, is actually a compromise between the strictly object/method model used in most of our discussion and the alternative that uses a single Event method. The underlying MS Windows architecture supports only a single event stream because it was not designed with C++ in mind. When models, the windowing system, the menu system, or any other process sends an event to a subclass of CView, it uses the same event-dispatching method. The Visual C++ environment, however, provides a structure for mapping various events directly to methods defined on the particular view subclass. This "message-mapping" technique retains most of the flexibility of the underlying MS Windows event model while still allowing programmers to exploit the messaging mechanisms in C++. The message-mapping technique will map input events onto C++ methods in much the way we designed our example controller implementation.

There is, however, only one OnUpdate method on any view. The OnUpdate method is invoked whenever some portion of the data displayed by the view has changed. The default implementation for OnUpdate is to damage, or invalidate, the entire window. This, of course, is always correct but may be inefficient. The OnUpdate method has two parameters, a long integer and a CObject pointer. In C++, a long integer can be used for just about anything, and the CObject class is the superclass of many model objects. These two parameters allow additional information about what has changed to be sent with an OnUpdate message. This single change notification message eliminates the need for abstract classes like CircuitView, but it requires more care when programming.

In our example, we defined the following change notification methods:

ChangeChip(ChipNum)

ChangeWire(WireNum)

MoveChip(ChipNum, NewLocation)

We could map this design onto the OnUpdate method by assigning an integer code to each type of change and then defining a subclass of CObject that would hold the remaining information. This would allow implementation of our design in Visual C++ without much difficulty.

Finally, every view has a GetDocument method that provides access to the CDocument object that implements the model. In keeping with some of the ugliness of C++, this CDocument pointer must be cast to a pointer of the type of model actually being viewed. After such a cast has been performed, all of the model methods can be invoked by the view and controller methods.

5.7.2 CDocument

The CDocument class is the superclass for all models. Its most valuable function is to provide the UpdateAllViews method. Because all views are subclasses of CView, the CDocument class maintains a list of views and then invokes the OnUpdate method on each registered view. This means that subclasses of CDocument will not need to manage view registration.

In addition to CDocument, there is also a CObject class that can be used as the superclass of objects like Chip and Wire. By using the generic CObject class as the superclass, various facilities are provided in Visual C++ for managing lists of such objects. In particular, there are facilities for saving and loading documents and their objects from files and for other cutting and pasting operations.

5.8 Summary

Interactive programs are separated into models and views. A model implements the functional model of the application. A view translates the model into an appropriate image. When the windowing system receives a user input, it forwards the event to the appropriate controller method on the appropriate view. The controller collects user inputs until there is sufficient information to make a change to the model. The model then notifies all of its views whenever a significant change happens. In response to change notification, a view will damage appropriate regions of the display. The windowing system will collect all damaged regions and will then invoke the Redraw methods of appropriate views. Each Redraw method will consult the model that the particular view is displaying in order to correctly redraw the damaged portion of the screen.