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

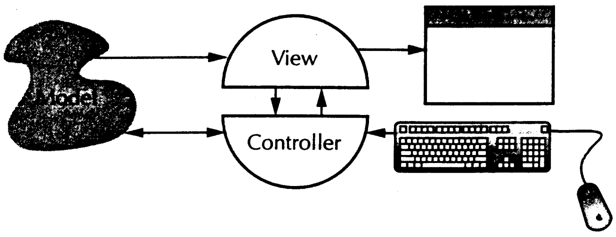

Figure 5-2 Model-view-controller

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 In some systems, such regions are called invalid or out of date.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

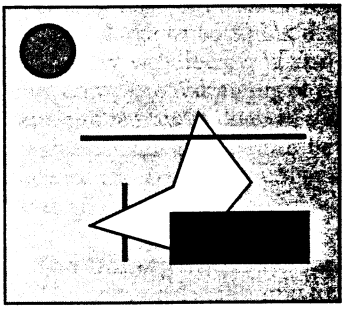

Figure 5-3 Shapes to be manipulated

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

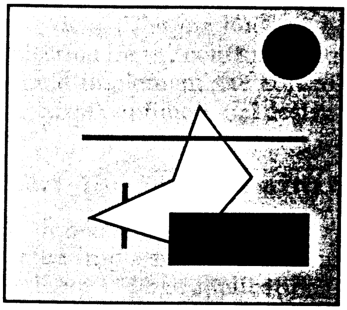

Figure 5-4 Erasing and redrawing the circle

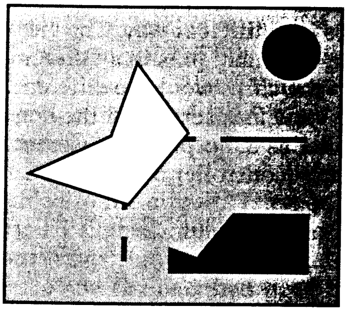

Figure 5-5 Erasing and redrawing the polygon

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

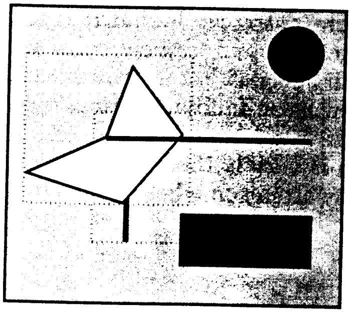

Figure 5-6 Damage/Redraw method showing damaged regions

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.

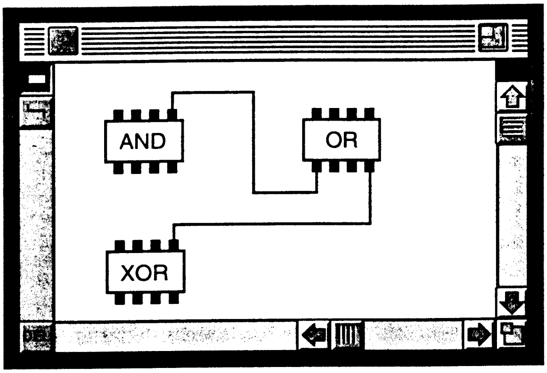

Figure 5-7 Dragging a chip

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.

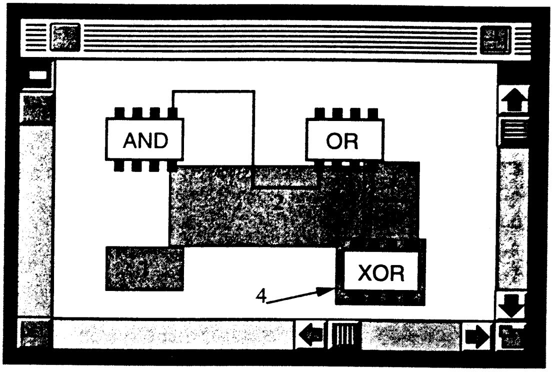

Figure 5-8 Damage regions to move a chip

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
```