11

Dialog Design

In chapter 8 we discussed the role of consistent look and feel. Tools like interaction design environments and other user interface layout tools are excellent for working through the look of an interface. Even paper and pencil or drawing on a whiteboard are excellent techniques for a design team to work through how a new user interface should look. This helps us to understand clearly what we are trying to implement in our view. The design of the feel, the sequence of inputs through which a user causes an action, is much more problematic because it is not visual. Implementing the controller code is difficult and requires tools and notations of its own.

For many widgets such as buttons, radio-boxes and check buttons the input sequence is trivial. The mouse goes down, the button on the screen looks pressed, the mouse goes up and the button pops out while some programmatic action is executed. Even a button is not as trivial as it looks. What happens if the mouse moves outside of the button's rectangle before the mouse is released? What happens if the mouse moves into the button while the mouse is already down? The on-screen button is one of the simplest of all input sequences, but there are still issues that are not immediately obvious and must be worked out.

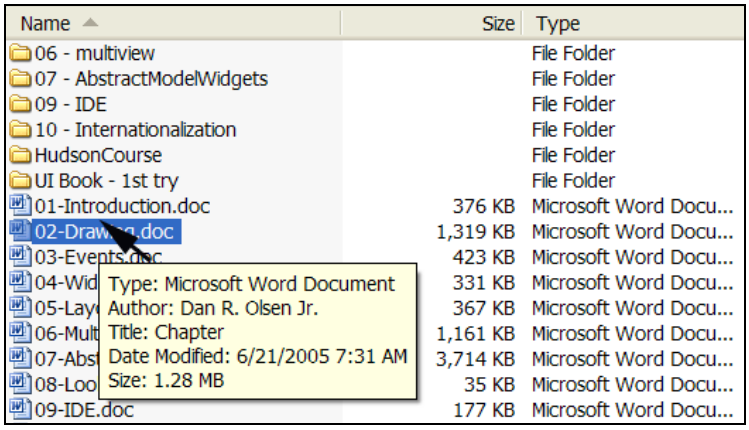


Figure 11.1 – Interacting with a file view

If we move beyond simple button interaction to the file folder view shown in figure 11.1 we will see many more complex issues. If the mouse hovers over a file for a period of time then the tip window should be shown. If the mouse clicks, then the file or folder should be selected. If the mouse double clicks then the file should be opened if possible. If the mouse clicks and then clicks again at some later time, the title of the file/folder should be presented for editing and renaming. If keyboard events occur, they should be ignored except when the dialog is in name editing mode. If "Enter" is pressed, then the file should be renamed, unless there is a problem with the name in which case the user should be notified and the renaming aborted. The event sequence mouse-down, mouse-move, mouse-up generally means that the file or folder is engaged in a drag and drop sequence to move it to a different location in the folder hierarchy. However, if the interface is in file rename mode, then that same sequence selects a portion of the name for deletion. These complexities grow in drawing or visualization environments where the simple click and drag techniques take on different meanings in a variety of situations.

One is at first inclined to resolve these issues in a series of prototypes. The problem is that the complexities of the code can frequently produce a prototype that has the right feel but nobody any longer knows what it actually does. The patches upon patches have buried the true behavior beyond recognition. The value of pencil and paper for design the look of an interface lies in the fact that it can be readily understood and openly discussed by all members of the design team. This is not true of 400 lines of C++ code. It is even less true of that same 400 lines scattered throughout the event handlers of Visual Basic.

Programs are a poor mechanism for designing input dialog for three reasons. The first is that the code is generally scattered among many different event handling methods. For example, the file drag and drop sequence must be divided among the mouseUp, mouseMove, mouseEnter, mouseExit and mouseUp event handler methods. Following a coherent trail of action becomes rather difficult. The second problem is that code is not expressively dense. That is, a 400 line implementation of an input dialog is very difficult to read, understand or discuss. Only 10-20% is visible at any one time. For a group design effort, the dialog design must fit on a whiteboard so that all can see and comment. The third reason is that once a dialog is scattered among so many methods it is difficult to reliably change during the ebb and flow of a design process. What we need is a compact, easily expressed, easily modified notation for capturing dialog design. That is the subject of this chapter.

We will look at three such notations. The first are mouse event diagrams that capture a single input task involving the mouse. The second are state machine diagrams. These have long been used for dialog design, yet they have serious practical drawbacks. Lastly we will look at propositional production systems that have the same expressive power as state machines without most of the problems.

Mouse event diagrams

In most cases we can characterize a task by user-caused events, the path of the mouse, the state of the mouse buttons and the screen objects beneath the mouse. Figure 11.2 shows a mouse event diagram for the simple pressing of an on-screen push button. The events are identified on the path by letters and then described below. The mouse buttons are shown white when that button is up and dark when that button is down. While the button is down and the mouse is dragging, the path is shown with a thicker line.

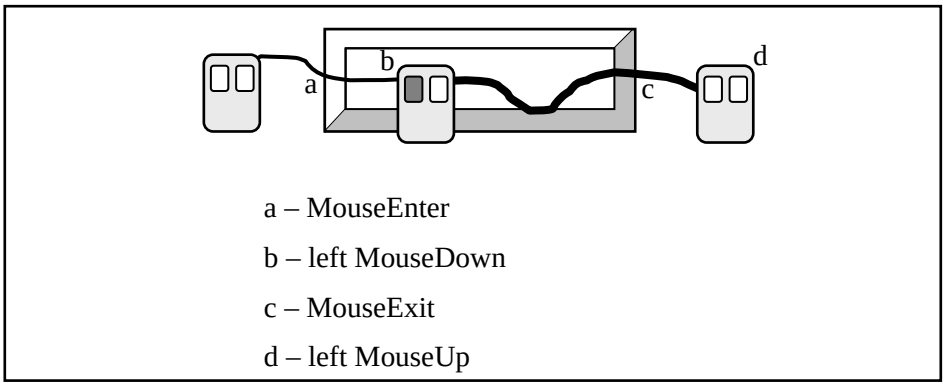


Figure 11.2 – Mouse event diagram for a push button

The condition being described in figure 11.2 is where the mouse button goes down inside the screen button and then is moved outside of the screen button while the left button is still down and then released outside of the button. This is not the traditional click behavior, but it must be addressed in a good button dialog design. The problem with the presentation in figure 11.2 is that it does not show what should happen to the screen button while all of this is going on. Figure 11.3 shows a modified version of figure 11.2 with the state of the button being presented at every event. This visual depiction at each event gives us a clear picture over time of what is to happen to the screen button during this sequence.

There are some interactive tasks that do not lend themselves as well to this presentation because multiple events happen in a particular place, such as clicking on this button. These can be handled by showing multiple event letters in the same position and then describing what should happen at each. The important issue is that a diagram like this clearly and briefly shows what should be happening during this interactive task.

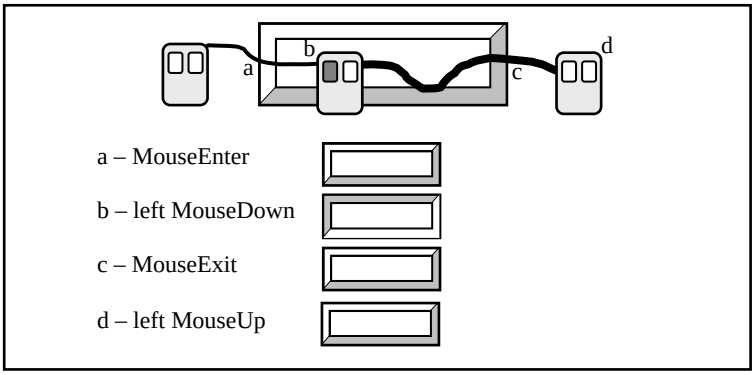


Figure 11.3 – Events augmented with widget presentation changes

Figure 11.4 shows a slightly more complicated task where the user is dragging the slider of a scroll-bar. When the mouse enters the scroll-bar the slider lightens to show that it is active. When the mouse goes down on the slider its border gets thicker to show that it is being moved. As the mouse leaves and reenters the region of the scroll-bar the slider stays active and moves with it. When the mouse is released, the slider reduces its border but stays light to show that it is responding to the mouse location.

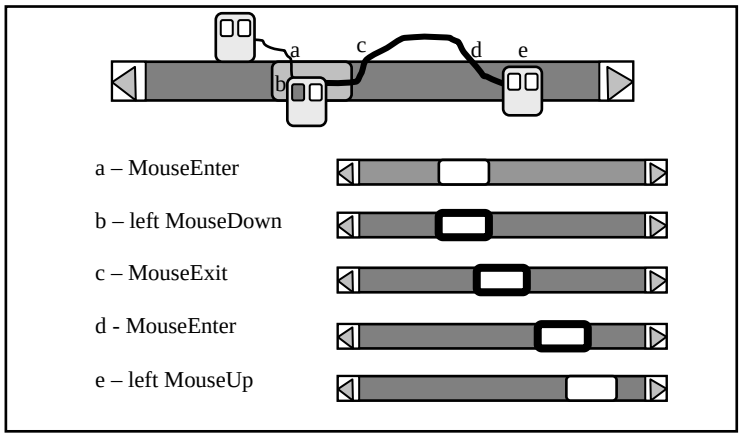


Figure 11.4 – Scroll-bar event diagram

If this diagram had been drawn on a whiteboard during a design meeting it would be very easy to discuss the behavior and whether it is appropriate. Perhaps some added texture would be appropriate rather than the thickened border. Perhaps the body of the scroll-bar should lighten rather than the slider to show that the scroll-bar is active. The point is that all of these options can easily be discussed, modified, remodified, modified again and finally documented in a way that everyone clearly understands.

For any particular widget there are many such tasks to be developed. In the case of our scroll-bar there are tasks for clicking in the arrows, clicking in the body, holding down the mouse in the arrows. We may want a different highlight when the mouse enters the body rather than the slider or an arrow. Using diagrams for each of these tasks, we can easily design and document all of these alternatives.

State representations of dialog

The mouse event diagrams are suitable for outlining a single interactive task. A real widget design will have many such tasks. However, the widget implementation must combine all of those tasks into the same event handling code. The event handling for push-buttons and scroll-bars is quite simple in comparison to many interactive tasks.

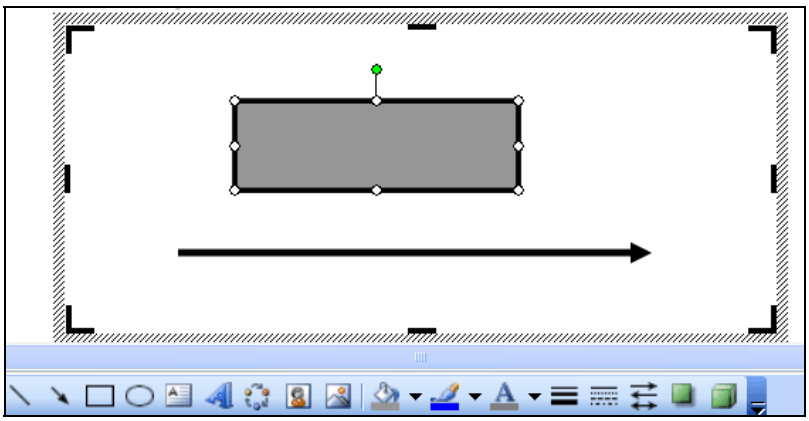


Figure 11.5 – Drawing program

Figure 11.5 shows a screen capture of the drawing facility in Microsoft Word. When the mouse goes down there are many things that could happen. If it goes down on the dot at the end of the control handle on the rectangle, then a rotation action begins. If it goes down on any of the corner or side dots, then a resizing action begins. If it goes down in open space, then (depending upon the current menu selections) a selection or the creation of a new object will begin. If it goes down on the black arrow then the rectangle is unselected, the arrow is selected and a drag of the arrow may or may not begin depending on how far the mouse moves. There are very many interactive tasks, with all of their behavior embedded in the mouseDown(), mouseMove() and mouseUp() event methods.

One of the first attempts to address this event complexity was through state machines[1] and state machine diagrams[2]. Figure 11.6 shows a diagram for the interactive behavior of the screen button design in figure 11.3. The circles are states and the transitions are *event/action* pairs. When an event is received that corresponds to a transition from the current state, the action for that transition is taken and the current state becomes the new state at the other end of the transition.

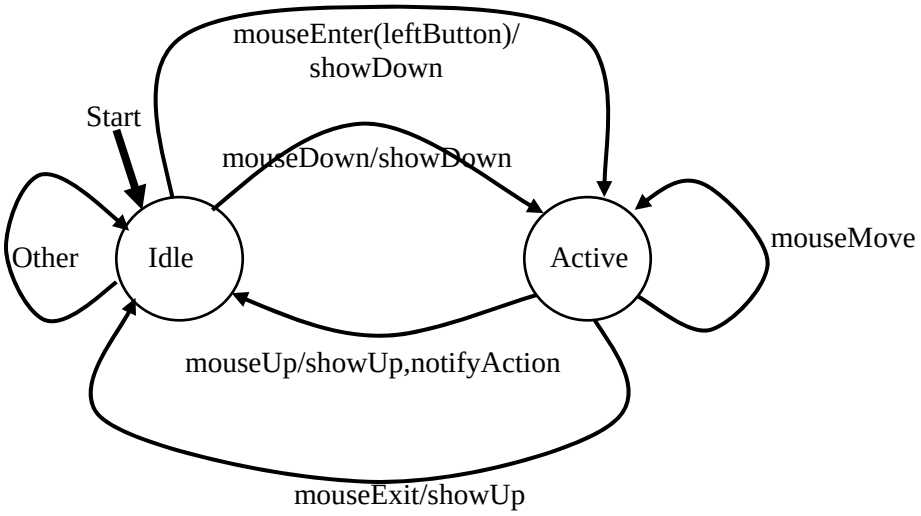


Figure 11.6 – State machine diagram

The state machine diagram in figure 11.6 illustrates both the advantages and disadvantages of this notation. It is quite easy to follow what events are acceptable and what actions should be taken on those events. Such diagrams are also relatively easy to translate into code. Each widget has a data member called `state` and there is an enumerated type for all of the states. Within each method there is a `switch` statement on the `state` variable and another on the input type to select the correct transition for that input event. In each case of the switch, the transition action is performed and `state` is set to the value for the next state. The problems are primarily ones of scale. If the number of states + transitions gets larger than 20-30 a single page is not sufficient to hold the information. In many situations the state machine is not a planar graph, which means that it is not possible to draw the diagram without transitions crossing each other. This makes it very complicated to read such a diagram.

An additional problem is the creation of tools for manipulating the diagrams. Systems like Visio[3] that can edit connected diagrams can help, but the tools are still rather uncomfortable.

A final problem with state machine diagrams is their expressive power. The problem lies in many of the combinations and repetition required to adapt the finite state machine to diverse dialog design needs. There have been a variety of adaptations including the use of context free grammars[4], and parser generation

technology[5]. None of these has done real well. The problem is that they do not sufficiently represent the intrinsic parallelism found in many dialogs. In many cases there are a variety of things to be specified but the order is only important in short chunks and there are many ways that a user might interleave their activities. Highly ordered dialogs are difficult for users to learn because they must map their intentions onto the prescribed order of the dialog. This leads to a serious gulf of execution problem.

To illustrate this difficulty, consider the problem of entering 8 pieces of information (A through H) each exactly once but in any order. To get this right the dialog must remember whether or not each piece of information has been received. Thus for each input we need one bit (received or not) in our state. For all 8 inputs we need 8 bits. This is a trivial amount of memory but the problem is difficult to express in a state machine diagram because there are 2^8 or 256 states. Consider trying to scale diagram 11.6 up to 256 circles with a similar number of labeled arrows. The problem is that whether or not input B has been received is completely independent of whether any other input has been received, yet the state diagram approach forces the designer to consider all possible combinations.

Propositional Production Systems

Propositional Production Systems (PPS)[6] are an alternative representation of finite state machines. In formal language theory they still belong to the same finite state category in the Chomsky hierarchy. However, they can express many problems in a linear number of statements rather than the exponential number that we saw with state diagrams. They are also edited using any text editor and much more spatially compact so that we can get more of the dialog on a whiteboard, piece of paper or computer screen. Because a PPS is still a finite state machine they also computationally tractable for performing automatic analysis. It is easy to design algorithms that automatically prove[7] statements such as “it is always possible to request help”, “action Z will never be performed without the user first entering a valid password”, “action P can always be reached in less than 5 steps from any point in the dialog”. Because of these properties, we will use PPS as our notation for dialog design. In most cases the primary value of the PPS notation is to work out dialog designs in a more understandable and modifiable form than code.

Field definition

One of the key concepts in a PPS is that the state space is divided up into fields. Each field has two or more conditions. Within a field the conditions are

mutually exclusive (the field can only hold one value at a time) but different fields are independent of each other. Formally there are some state dependencies that cannot be conveniently represented, but they rarely occur in practice. Figure 11.7 shows the field definition for our 8-input problem. There are only 8 fields and 16 conditions rather than the 256 states required in state diagrams. In this machine the conditions Ayes and Ano cannot hold at the same time but they are independent of all other conditions.

```
A { Ayes, Ano }
B ( Byes, Bno }
C { Cyes, Cno }
D { Dyes, Dno }
E { Eyes, Eno }
F { Fyes, Fno }
G { Gyes, Gno }
H { Hyes, Hno }
```

Figure 11.7 – PPS fields for the 8-input problem

For convenience we define a variety of types of fields with their conditions to actually model interactive dialogs. In this book we will use a notation standard for each type of condition. It helps in reading the dialog specification and in translating the specification into code. The types of fields are:

- Input events (* before the condition name)
  {*mouseDown, *mouseUp, *mouseMove}
- Input modifiers (* after the condition name)
  LeftMouse{leftUp*, leftDown* }
  RightMouse {rightUp*, rightDown*}
  ControlKey {controlUp*, controlDown*}
  ShiftKey {shiftUp*, shiftDown* }
- State information to be remembered
  SliderDisplay { sliderInactive, sliderActive }
  ButtonState { buttonUp, buttonDepressed }
  MouseDragging { idle, dragging }
- Query fields (? before name)
  Password {?validPassword, ?noPassword }
  ScrollPos {?stepUp, ?pageUp, ?slider, ?pageDwn, ?stepDwn }
- Actions (! before name, sometime there are parameters)
  {!drawButtonUp, !drawButtonDown}
  {!notifyButtonAction}
  {!increment(amount), !decrement(amount), !scroll(newPos) }

There is only one input event field because only one event can be received by an application at a time. This is where one lists all of the input events that are relevant for a particular widget. The input modifiers are arranged in separate fields because they are independent of each other. It is possible to press the right mouse button while holding down the shift key and at the same time ignore whether the control key is pressed or not. We separate input modifiers from input events because modifiers by themselves do not cause actions to occur. We must know the modifier state in many cases so that we can take the correct action when one is required.

Most of the control of the interactive dialog is encoded in state fields. State fields can represent many different things. It is up to the dialog designer to decide how various pieces of information can be modeled. The examples later in the chapter will show how these are used.

Query fields are special in that their conditions are not remembered but rather the result of some other code. For example, it is not possible to encode all of the possible passwords in a PPS. It is far simpler to create a piece of code that looks up the password and determines if it is valid. We can define a method Password() that returns validPassword or noPassword. We can use the returned condition to then control the behavior of our controller implementation. We will use query fields for our essential geometry information.

Action fields are placeholders for pieces of code to be executed. We group them into fields because many actions are mutually exclusive. For example you do not want to increment the scroll-bar at the same time you are decrementing it. Actions are where the model methods are invoked. Frequently we need to provide parameters and we generally handle this informally in the notation.

Productions

The syntax of a PPS is encoded in a set of productions rather than in state transitions. Each production has a left-hand side (antecedent) and a right-hand side (consequent). If all of the conditions in the antecedent hold then the consequent is asserted. Input events, input modifiers and query conditions can only occur in the antecedent because they cannot be changed by the controller. Actions can only occur in the consequent. A PPS consists of multiple productions and all productions with matching antecedents fire at once. This is to prevent race conditions where one production can change the conditions for a later production. This parallel firing policy requires some care when converting a PPS into code as we will discuss later. Example rules might be:

```
*mouseUp, shiftDown*, selectClick -> !addSelect, selectModeIdle

*mouseUp, shiftUp*, selectClick -> !newSelect, selectModeIdle
```

These two productions only fire when the mouseUp event has occurred and the state of the controller is selectClick. They differentiate their behavior based on whether the shift key is pressed.

Scroll-bar dialog example

Our first step in defining a PPS specification is to identify the interactive tasks that the widget must accomplish. For the scroll-bar example in figure 11.4 we can identify the following.

1. Click in the left arrow to step left
2. Click in the right arrow to step right
3. Click in the left body area to page left
4. Click in the right body area to page right.
5. Drag the slider to scroll

Based on these tasks we can identify a set of fields that have information we will need in our dialog. In figure 11.8 there is a field with all of the events we will receive from the mouse. There is a query field for the essential geometry. We have a State field that remembers what we are doing from event to event. To know the right thing to do on mouse-up we must remember what we were doing on mouse-down. Our actions have been grouped into two different fields, each handling a different set of issues. The Model field handles the actual changes to the model, the Feedback field handles highlighting the scroll-bar to provide user feedback. We separate these actions because between the groups they are not mutually exclusive. There may be rules that perform a model action at the same time that they perform a feedback action, which is just fine. We do not, however, want to !pageLeft and !pageRight at the same time. Putting these two actions in the same field specifies that they are mutually exclusive.

```
Input {*mouseDown, *mouseMove, *mouseUp }
      // events we will receive as the mouse moves
State { idle, steppingLeft, steppingRight, pagingLeft, pagingRight, dragging }
      // remembers what the controller is doing from one event to another
EssentialGeometry { ?leftArrow, ?rightArrow, ?leftBody, ?rightBody, ?slider }
      // query field that identifies which of the 5 regions the mouse is in at the current time.
Model { !stepLeft, !stepRight, !pageLeft, !pageRight, !dragStart, !dragEnd, !dragScroll }
      // action methods that actually do the various things that a scrollbar should do
Feedback {  !sliderActive, !leftArrowActive, !rightArrowActive, !bodyActive, !allPassive }
      // action methods to handle highlighting the slider
```

Figure 11.8 – Fields for scroll-bar PPS.

We can build up our set of production rules by looking at each task in turn. Tasks 1-4 involve simple clicking in regions. We can handle these with the rules in figure 11.9. Each task has two rules, one for mouse-down and one for mouse-up. Each rule works with the State and EssentialGeometry fields to decide what should be done.

```
1.  *mouseDown, idle, ?leftArrow -> steppingLeft, !leftArrowActive
2.  *mouseUp, steppingLeft -> idle, !allPassive, !stepLeft
3.  *mouseDown, idle, ?rightArrow -> steppingRight, !rightArrowActive
4.  *mouseUp, steppingRight -> idle, !allPassive, !stepRight
5.  *mouseDown, idle, ?leftBody -> pagingLeft, !bodyActive
6.  *mouseUp, pagingLeft -> idle, !allPassive, !pageLeft
7.  *mouseDown, idle, ?rightBody -> pagingRight, !bodyActive
8.  *mouseUp, pagingRight -> !allPassive, !pageRight
```

Figure 11.9 – Production rules for tasks 1-4

Looking at our rules, we realize that we are not accounting for mouse movement in these tasks. The design team discussion might be:

"What happens on mouse move?"

"No problem, these are all clicking tasks."

"Yes but what if the mouse moves out of the left arrow before mouse up?"

"Well I guess we should unhiglight the arrow."

"And what happens if the mouse movement goes completely outside of the scroll-bar rectangle?"

"We would still unhighlight the arrow."

"But what if the mouse is still down and the mouse is moved back?"

"I think the arrow becomes active again"

"That doesn't feel like a click task to me"

The advantage here is that we can have a discussion about 8 rules and 5 fields in a way that is not possible with several pages of code. Having discussed the alternatives, the input field is changed to that shown in figure 11.10 and the production rules are changed to those shown in figure 11.11. Notice in rules 2, 5, 8, 11 and 13 of figure 11.11 we have added the "~" to indicate that the condition must not hold for the rule to fire. This is simpler than a separate rule for each of the other conditions. We will need to remember this case, however, when we translate into code. Rule 13 handles all of the mouse exit issues.

```
Input {*mouseDown, *mouseMove, *mouseUp, *mouseExit}
      // events we will receive as the mouse moves
```

Figure 11.10 – Modified Input field

1. *mouseDown, idle, ?leftArrow -> steppingLeft, !leftArrowActive
2. ***mouseMove, steppingLeft, ~?leftArrow -> !allPassive, idle**
3. *mouseUp, steppingLeft -> idle, !allPassive, !stepLeft
4. *mouseDown, idle, ?rightArrow -> steppingRight, !rightArrowActive
5. ***mouseMove, steppingRight, ~?rightArrow -> !allPassive, idle**
6. *mouseUp, steppingRight -> idle, !allPassive, !stepRight
7. *mouseDown, idle, ?leftBody -> pagingLeft, !bodyActive
8. ***mouseMove, pagingLeft, ~?leftBody -> !allPassive, idle**
9. *mouseUp, pagingLeft -> idle, !allPassive, !pageLeft
10. *mouseDown, idle, ?rightBody -> pagingRight, !bodyActive
11. ***mouseMove, pagingRight, ~?rightBody -> !allPassive, idle**
12. *mouseUp, pagingRight -> !allPassive, !pageRight
13. ***mouseExit ~idle -> !allPassive idle**

Figure 11.11 – Updated rules to handle mouse movement

We still have not addressed task 5 where the slider is being dragged. We can address that task with the rules in figure 11.12. While discussing these rules we discovered that we needed to handle the mouse-focus in case the user strays outside the box. To take care of these we created the additional action field shown in figure 11.13.

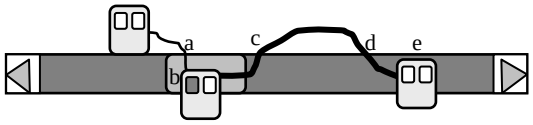
14. *mouseDown, idle, ?slider -> !sliderActive, dragging, !dragStart(mousePoint), !getMouseFocus
15. *mouseMove, dragging -> !dragScroll(mousePoint)
16. *mouseUp, dragging -> !dragEnd(mousePoint), idle, !releaseMouseFocus

Figure 11.12 – Additional rules to handle mouse dragging

Focus {!getMouseFocus, !releaseMouseFocus }

Figure 11.13 – additional action field to handle mouse focus

We now have a PPS that we believe captures the entire controller dialog for a scroll-bar. Before jumping into code we want to first check our PPS to see if it is functioning correctly. We do this by working through each of our interactive tasks for which we created mouse event diagrams. We want to see if our PPS does the right thing for each of these tasks. In figure 11.14 we work through our event diagram identifying what will happen to the state variables, what rules will fire and what actions will be taken.



start – State{idle}
a – *mouseEnter – no rule fires, nothing happens
b – *mouseDown – rule 14 fires, !sliderActive, State{dragging}
!dragStart(mousePoint), !getMouseFocus
* - *mouseMove - rule 18 repeatedly fires calling !dragScroll()
c – *mouseExit – rule 13 fires and does the wrong thing –
we correct rule 13 to those shown in figure 11.15 and now
no rule fires, which is correct
d - *mouseEnter – no rule fires, nothing happens
e – *mouseUp – rule 19 fires, !dragEnd, State{idle},
!releaseMouseFocus

Figure 11.14 – Working through PPS with mouse event diagram

As we are working through the rules on this interactive task we find that rule 13 is too general in that it shuts down the scroll bar even though we have the mouse focus. We therefore replace rule 13 with rules 13-16 in figure 11.15 and continue our work. We also realize from our mouse event diagram that we need to account for the fact that only left mouse button should trigger any of the actions. We add an additional input field to create the final field list shown in figure 11.16 and we update the rules to account for the button information as shown in figure 11.15.

1. *mouseDown, idle, **leftDown***, ?leftArrow -> steppingLeft, !leftArrowActive
2. *mouseMove, steppingLeft, ~?leftArrow -> !allPassive, idle
3. *mouseUp, **leftUp***, steppingLeft -> idle, !allPassive, !stepLeft
4. *mouseDown, idle, **leftDown***, ?rightArrow -> steppingRight, !rightArrowActive
5. *mouseMove, steppingRight, ~?rightArrow -> !allPassive, idle
6. *mouseUp, **leftUp***, steppingRight -> idle, !allPassive, !stepRight
7. *mouseDown, idle, **leftDown***,?leftBody -> pagingLeft, !bodyActive
8. *mouseMove, pagingLeft, ~?leftBody -> !allPassive, idle
9. *mouseUp, **leftUp***, pagingLeft -> idle, !allPassive, !pageLeft
10. *mouseDown, idle, **leftDown***, ?rightBody -> pagingRight, !bodyActive
11. *mouseMove, pagingRight, ~?rightBody -> !allPassive, idle
12. *mouseUp, **leftUp***, pagingRight -> !allPassive, !pageRight
13. ***mouseExit steppingLeft -> !allPassive idle**
14. ***mouseExit steppingRight -> !allPassive idle**
15. ***mouseExit pagingLeft -> !allPassive idle**
16. ***mouseExit pagingRight -> !allPassive idle**
17. *mouseDown, **leftDown***, idle, ?slider -> !sliderActive, dragging, !dragStart(mousePoint), !getMouseFocus
18. *mouseMove, dragging -> !dragScroll(mousePoint)
19. *mouseUp, **leftUp***, dragging -> !dragEnd(mousePoint), idle, !releaseMouseFocus

Figure 11.15 – Revised production rules

```
Input {*mouseDown, *mouseMove, *mouseUp }
      // events we will receive as the mouse moves
LeftButton {leftDown*, leftUp* }
      // state of the left mouse button
State { idle, steppingLeft, steppingRight, pagingLeft, pagingRight, dragging }
      // remembers what the controller is doing from one event to another
EssentialGeometry { ?leftArrow, ?rightArrow, ?leftBody, ?rightBody, ?slider }
      // query field that identifies which of the 5 regions the mouse is in at the current time.
Model { !stepLeft, !stepRight, !pageLeft, !pageRight, !dragStart, !dragEnd, !dragScroll }
      // action methods that actually do the various things that a scrollbar should do
Feedback {  !sliderActive, !leftArrowActive, !rightArrowActive, !bodyActive, !allPassive }
      // action methods to handle highlighting the slider
Focus {!getMouseFocus, !releaseMouseFocus }
      // manage the mouse focus
```

Figure 11.16 – Revised PPS fields for scroll-bar

Translating PPS into code

Having gone through the design process we now have a PPS specification in figures 11.15 and 11.16 that we believe accounts for everything that our scroll-bar's controller will need and we have corrected several design and specification errors. We are now ready to translate the PPS into controller code. In this step it is important to follow a principled process step by step so that we accurately

translate the rules into code. We may also find additional errors in the specification as we go through this process and find rules that are in conflict with each other.

In translating a PPS into code we are assuming that the controller is part of a view class. The controller is implemented primarily as event handling methods on such a class.

Encoding of fields

The first step is to translate all of our fields into code. The input events are translated into event handler methods. There is one such method for each of the input events used in the PPS. The naming of these methods will depend upon the tool-kit platform on which the controller is being built. The input modifiers are not directly encoded in the class. It is normal for every input event method to have a parameter that contains the event information including the mouse location and the state of various buttons. There are methods on such event objects for determining the state of the control or shift keys for example. Where those input conditions occur in the production rules we will replace them with code appropriate to the underlying platform.

State fields are very important because they are our memory from event to event. For every state field we create an enumeration of all of the conditions for that field and a data member of that type. Figure 11.17 shows Java code for our scroll-bar's state field. Note that many controller designs will have more than one state field. Our scroll-bar only required one for its implementation. The data member state has also been initialized to IDLE as its starting state.

```
private enum State {
        IDLE, STEPPING_LEFT, STEPPING_RIGHT,
        PAGING_LEFT, PAGING_RIGHT, DRAGGING }
private State state = IDLE;
```

Figure 11.17 – Implementing the State field

Query fields also have an enumerated type, but their values are dynamically computed rather than stored in a variable. Figure 11.18 shows how our scroll-bar's EssentialGeometry field is implemented.

```
private enum EssentialGeometry{
        LEFT_ARROW, RIGHT_ARROW, LEFT_BODY, RIGHT_BODY, SLIDER };
private EssentialGeometry essentialGeometry(int mouseX, mouseY)
{
    . . . . code to determine which region of the scroll bar has been selected . . .
}
```

Figure 11.18 – Implementing the EssentialGeometry Query field

The action fields and their conditions are implemented as a method for each action condition. These methods will be invoked whenever they appear in a rule that is to fire.

To tidy up our PPS we want to define an ordering on the fields. We want them ordered in the sequence that we will check them. Our first field should be the input events followed by the input modifier fields. This is because our first task will be to determine what the user has input. Secondly we order our state fields. It is helpful to place the most discriminatory fields first. These are the ones that are most frequently used to decide what action to take. This is a judgment call on the part of the designer. After the state fields come the query fields. We place these later because they involve more complex code than simple field checking and we only want to call them when necessary. Lastly we place the action fields because we will invoke these after everything else. The order is:

- Input fields
- Input modifier fields
- State fields in most used to least used order
- Query fields
- Action fields

It is helpful to get in the habit of using this order from the beginning as was done in figure 11.16.

Reorder rules

During the design phase we arranged our rules in the order that made sense as we worked through our interactive tasks. This ordering is not appropriate for the code because the code is organized by input event method rather than by interactive task. We also need to reorder the antecedents (left-hand side) in the same order as we have ordered the fields. These two reorderings will greatly simplify the step-by-step construction of code that accurately implements the PPS. The reordered rules for the scroll-bar are found in figure 11.19.

```
1.  *mouseDown, leftDown*, idle, ?leftArrow -> steppingLeft, !leftArrowActive
2.  *mouseDown, leftDown*, idle, ?rightArrow -> steppingRight, !rightArrowActive
3.  *mouseDown, leftDown*, idle,?leftBody -> pagingLeft, !bodyActive
4.  *mouseDown, leftDown*, idle,?rightBody -> pagingRight, !bodyActive
5.  *mouseDown, leftDown*, idle, ?slider -> !sliderActive, dragging, !dragStart(mousePoint),
        !getMouseFocus

6.  *mouseMove, steppingLeft, ~?leftArrow -> !allPassive, idle
7.  *mouseMove, steppingRight, ~?rightArrow -> !allPassive, idle
8.  *mouseMove, pagingLeft, ~?leftBody -> !allPassive, idle
9.  *mouseMove, pagingRight, ~?rightBody -> !allPassive, idle
10. *mouseMove, dragging -> !dragScroll(mousePoint)

11. *mouseUp, leftUp*, steppingLeft -> idle, !allPassive, !stepLeft
12. *mouseUp, leftUp*, steppingRight -> idle, !allPassive, !stepRight
13. *mouseUp, leftUp*, pagingLeft -> idle, !allPassive, !pageLeft
14. *mouseUp, leftUp*, pagingRight -> !allPassive, !pageRight
15. *mouseUp, leftUp*, dragging -> !dragEnd(mousePoint), idle, !releaseMouseFocus

16. *mouseExit steppingLeft -> !allPassive idle
17. *mouseExit steppingRight -> !allPassive idle
18. *mouseExit pagingLeft -> !allPassive idle
19. *mouseExit pagingRight -> !allPassive idle
```

Figure 11.19 – Reordered scroll-bar rules

Group rules into event methods

This reordering of the rules places all of the rules for a given input method together and we can group them accordingly. From all over our design we now have all of the rules that might fire on a *mouseDown or *mouseExit event gathered together. We can now start building our event method implementations as shown in figure 11.20. Each rule is grouped into an event method by number. It is a waste to recopy the rules because will be making continuous changes to these methods as our process proceeds. Remember that the names of these methods and their parameters are determined by our tool-kit's event handling protocol. For example, some systems group mouse events into the same input event and distinguish up or down by the button modifiers.

```
public void mouseDown (MouseEvent e)
{     // rules 1, 2, 3, 4, 5
}
public void mouseMove (MouseMoveEvent e)
{     // rules 6, 7, 8, 9, 10
}
public void mouseUp(MouseEvent e)
{     // rules 11, 12, 13, 14, 15
}
public void mouseExit(MouseEvent e)
{     // rules 16, 17, 18, 19
}
```

Figure 11.20 – Event handler methods with their PPS rules

Translate the antecedent tests

Within each method there are several rules and we must translate the antecedents of the rules into tests using if or switch statements. We can start first with the mouseDown() event. The first condition of every rule is *leftDown. We translate this into an if test as shown in figure 11.21. The code is not drawn from any particular tool-kit and would be similar in most tool-kits.

```
public void mouseDown(MouseEvent e)
{
     if (e.leftButtonState == BUTTON_DOWN)
     {     // rules 1, 2, 3, 4, 5
     }
}
```

Figure 11.21 – Translating the *leftDown input

Looking further at rules 1-5 we see that every rule is conditioned on the state being idle. We further translate the mouseDown() method as shown in figure 11.22. Note that just because all rules have the same condition we cannot ignore that condition because there is the implicit alternative to take no action. Note also that by ordering the fields and then reordering the antecedents in the same order we have simplified our translation process.

```
public void mouseDown(MouseEvent e)
{
     if (e.leftButtonState == BUTTON_DOWN)
     {
          if (state == IDLE)
          {     // rules 1, 2, 3, 4, 5
          }
     }
}
```

Figure 11.22 – Translating the state field

The next field to consider in the mouseDown() method is the essentialGeometry() query field. Because there are many alternatives in this case, the best option is a switch statement as shown in figure 11.23.

```
public void mouseDown(MouseEvent e)
{
    if (e.leftButtonState == BUTTON_DOWN)
    {
        if (state == IDLE)
        {
            switch( essentialGeometry(e.mouseX, e.mouseY) )
            {
            case LEFT_ARROW:  // rule 1
                break;
            case RIGHT_ARROW: // rule 2
                break;
            case LEFT_BODY: // rule 3
                break;
            case RIGHT_BODY: // rule 4
                break;
            case SLIDER: // rule 5
                break;
            }
        }
    }
}
```

Figure 11.23 – Translating the essentialGeometry field

We now have every rule for mouseDown() in its own group by itself. We therefore know that only that rule can fire and we are free to translate its consequent into action code. In some situations we may have exhausted all of the antecedent conditions and still have more than one rule in the group. At this point we check the consequents of each rule to see if they are in conflict. Two consequents are in conflict if they contain two different conditions for the same field. We cannot assert two mutually exclusive conditions. If there is a conflict, then there is an error in our specification and we need to figure out what it is and correct it. This is a major reason to pursue a careful step-by-step translation so that we flush out all such problems. If there are no conflicts then we can translate the consequents of the rules in any order we choose.

Our consequent should only contain state conditions or action conditions. State conditions are translated as assignments and action conditions are translated as method calls. This is shown for mouseDown() in figure 11.24.

```
public void mouseDown(MouseEvent e)
{
    if (e.leftButtonState == BUTTON_DOWN)
    {
        if (state == IDLE)
        {
            switch( essentialGeometry(e.mouseX, e.mouseY) )
            {
            case LEFT_ARROW:  // rule 1
                state=STEPPING_LEFT;
                leftArrowActive();
                break;
            case RIGHT_ARROW: // rule 2
                state=STEPPING_RIGHT;
                rightArrowActive();
                break;
            case LEFT_BODY: // rule 3
                state=PAGING_LEFT;
                bodyActive();
                break;
            case RIGHT_BODY: // rule 4
                state=PAGING_RIGHT;
                bodyActive();
                break;
            case SLIDER: // rule 5
                sliderActive();
                state=DRAGGING;
                dragStart(e.mouseX, e.mouseY);
                getMouseFocus();
                break;
            }
        }
    }
}
```

Figure 11.24 – Translating the rule consequents

There is one case that may arise while translating antecedents and that is when within a group of rules there are some rules that mention a particular field and some rules that do not. Using a field in a test is designed to separate groups of rules into sub-groups. If a rule does not mention a field being used for a decision, then it must be duplicated into all subgroups derived from that field.

Summary

To build the feel of a new widget into the controller we need a notation that lets us conveniently discuss and analyze our design before translating it into code. Mouse event diagrams are a useful means for working through sequences of events that may occur in particular interactive tasks. They capture not only the event sequence but also the changes to the visual appearance as events are

received. State machines have been used historically to capture valid event sequences, but they have problems in scaling up to realistic problems. Propositional Production Systems provide a convenient notation for capturing and analyzing input event and action handling. There is a straightforward technique for translating a PPS into working controller code.

Exercises

1. Given a drawing program such as Visio, Adobe Illustrator or pictures in Word, create a mouse event diagram for drawing a new line.
2. For the game MineSweeper, create a full mouse event diagram (with pictures for the states) for the task of selecting a cell.
3. What is the advantage of the PPS notation over simple state machines?
4. Develop a PPS specification for a text box. Include selections of a single point, a range of characters, normal text keys as well backspace and left/right arrows. Be sure to address essential geometry and semantic actions.
5. Translate the following PPS into appropriate code.

```
input {*md,*mm,*mu}
check { ?hit, ?miss }
status { idle, seeking, working }
control { !append, !delete, !reorder}
active { first, last }

first *md ?hit -> seeking
first *md ?miss -> working
first *mm -> last !reorder
last seeking *mu -> !delete
last idle *mm -> !append first
*mu -> idle
```

---

[1] Robert J. K. Jacob, "Using Formal Specifications in the Design of a Human-computer Interface," *Communications of the ACM*, v.26 n.4, p.259-264, April 1983

[2] Newman, W.M. "A System for interactive Graphical Programming." *Spring Joint Computer Conference*, AFIPS Press (May 1968)

[3] Walker, M. H., Eaton, N. *Microsoft Office Visio 2003 Inside Out*, Microsoft Press (2003).

[4] Hanau, P. R. and Lenorovitz, D. R. "Prototyping and Simulation Tools for User/Computer Dialogue Design" *Computer Graphics and Interactive Techniques (SIGGRAPH '80)* ACM Press, New York, NY, 271-278

[5] Olsen, D. R. and Dempsey, E. P. "SYNGRAPH: A Graphical User Interface Generator." *Computer Graphics and Interactive Techniques (SIGGRAPH '83)*. ACM Press, New York, NY, (1983) 43-50.

[6] Olsen, D. R. "Propositional Production Systems for Dialog Description." *Human Factors in Computing System (CHI '90)*. ACM (1990) 57-64.

[7] Olsen, D. R., Monk, A.F. and Curry, M.B, "Algorithms for Automatic Dialogue Analysis Using Propositional Production Systems," *Human Computer Interaction*, 10, 39-78 (1995).