# Object-Oriented Programming in Python

**Michael H. Goldwasser**

*Saint Louis University*

**David Letscher**

*Saint Louis University*

PEARSON
Prentice Hall

*Upper Saddle River, New Jersey 07458*

CHAPTER 15

# Event-Driven Programming

Up to this point, we would describe the style of our programs as flow driven. They proceed sequentially, executing one command after another until the end of the program is reached. Control structures might be used to more creatively express the desired sequence of operations, yet still the execution follows one continuous flow, finishing one command and immediately executing another.

In this chapter, we examine another very important style known as ***event-driven programming***. In this paradigm, an executing program waits passively for external ***events*** to occur, and then responds appropriately to those events. Consider for example, a modern word processor. Even when the software is running, it typically sits quietly unless the user of the program does something. When an event occurs, such as the selection of a menu item, the typing of characters, or the pressing of a button, the software reacts appropriately. Each event may be handled with a different response.

In coding how various events should be handled, we will use familiar Python programming techniques (e.g., control structures, functions, and objects). The major shift in thinking will involve the high-level design of a program and our view of how an executing program should proceed.

We focus primarily on ***graphical user interfaces (GUIs)***, which are classic examples of event-driven programming. The user's actions are viewed as events that trigger reactions from the software. When a particular type of event occurs, a method is executed to deal with this event. If multiple events occur then a method is initiated for each event. A web server is another classic example of event-driven software. The server simply waits until a request is received for information, only then working to properly handle the request. That request is considered an external event to the server, because it was triggered by some other entity. We will explore some examples of network programming in Chapter 16.

## 15.1 Basics of Event-Driven Programming

We have already seen a basic form of event-driven programming in our earlier endeavors. We use the **raw_input** function in order to get user input. Consider the following simple program:

```
print 'What is your name?',
name = raw_input( )                          # wait indefinitely for user response
print 'Hello %s. Nice to meet you.' % name
```

When the **raw_input** function is called, our flow of control comes to a halt, waiting indefinitely until the user enters a response. Once that user event occurs, we continue on our way. A similar example, introduced on page 115 of Chapter 3, uses the wait( ) method of the graphics library.

```
paper = Canvas( )
cue = paper.wait( )                          # wait indefinitely for user event
ball = Circle(10, cue.getMouseLocation( ))
ball.setFillColor('red')
paper.add(ball)
```

The paper.wait( ) call causes the execution of our program to be delayed until the user triggers an appropriate event on the canvas, such as a mouse click.

However, both of these examples are still sequential. The flow of control is expressed just as all of our other programs. One statement is executed after the next; we can simply view **raw_input**( ) as any other function call, although one that may take a while before returning. Unfortunately, waiting for events in this way is not very flexible. When writing a program, we have to forecast the precise opportunities for user interaction, and the user must follow this script.

In most software, the user has more freedom to control the actions of a program. At any point, a user has the option of selecting menu items, entering keyboard input, using a scroll bar, selecting text, and much more. For example, a simple drawing program might allow the user to click on one button to add a shape, on another button to delete a shape, to drag a shape with the mouse from one position to another, or to save an image through an appropriate menu selection or the correct keyboard shortcut. When implementing this type of software, a sequential train of thought is not as meaningful. Instead, the program is described through ***event handling***. The program declares various events that should be available to the user, and then provides explicit code that should be followed to handle each individual type of event when triggered. This piece of code is known as an ***event handler***.

### Event handlers

Often, a separate event handler is declared for each kind of event that can be triggered by a user interaction. An event handler is typically implemented either as a stand-alone function or as an instance of a specially defined class. When programmed as a stand-alone function, event handlers are known as ***callback functions***. The appropriate callback function is registered in advance as a handler for a particular kind of event. This is sometimes described as registering to ***listen*** for an event, and thus handlers are sometimes called ***listeners***. Each time such an event subsequently occurs, this function will be called.

With object-oriented programming, event handling is typically implemented through an event-handling class. An instance of such a class supports one or more member functions that will be called when an appropriate event occurs. The advantage of this technique over use of pure callback functions is that a handler can maintain state information to coordinate the responses for a series of events.

### The event loop

The design of event-driven software is quite different from our traditional flow-driven programming, although there is still a concept of the main flow of control. When the software first executes, initialization is performed in traditional fashion, perhaps to create and decorate one or more windows and set up appropriate menus. It is during this initialization that event handlers are declared and registered. Yet once initialization is complete, the execution may reach a point where the next task is simply to wait for the user to do something.

We want our software to be ready to handle any number of predefined events triggered in arbitrary order. This is typically accomplished by having the main flow of control enter what is known as an ***event loop***. This is essentially an infinite loop that does nothing. Yet when an event occurs, the loop stops to look for an appropriately registered handler, and if found that handler is called (otherwise the event is ignored). When a handler is called, the flow of control is temporarily ceded to the handler, which responds appropriately. Once the handler completes its task, the default continuation is to re-enter the event loop (although there are techniques to “quit” the event loop, if that is the appropriate consequence of a user action). Figure 15.1 provides a simple view of such an event-driven flow of control.

### Threading

There are several forms of event-driven programming. The first question at hand is what happens to the program’s flow of control once an event loop begins. In one model, the primary program cedes the flow of control to the event loop. Another model uses what is known as ***multithreaded*** programming, allowing the main program to continue executing, even while the event loop is monitoring and responding to events. The main routine and this event loop run simultaneously as separate ***threads*** of the program. Threading can be supported by the programming language and the underlying operating system. In reality, the threads are sharing the CPU, each given small alternating time slices in which to execute.

Just as we distinguish between the case of the event loop running in the primary thread (thus blocking the main program), or in an independent thread, we can ask how the event loop invokes a handler. In some models, the handler executes using the same thread as the event loop, thus blocking the loop’s monitoring of other events in the interim. However, it is possible to make further use of threads by having the event loop invoke each handler with an independent thread. In this way, the handler can continue to run even while the event loop continues listening for other events.

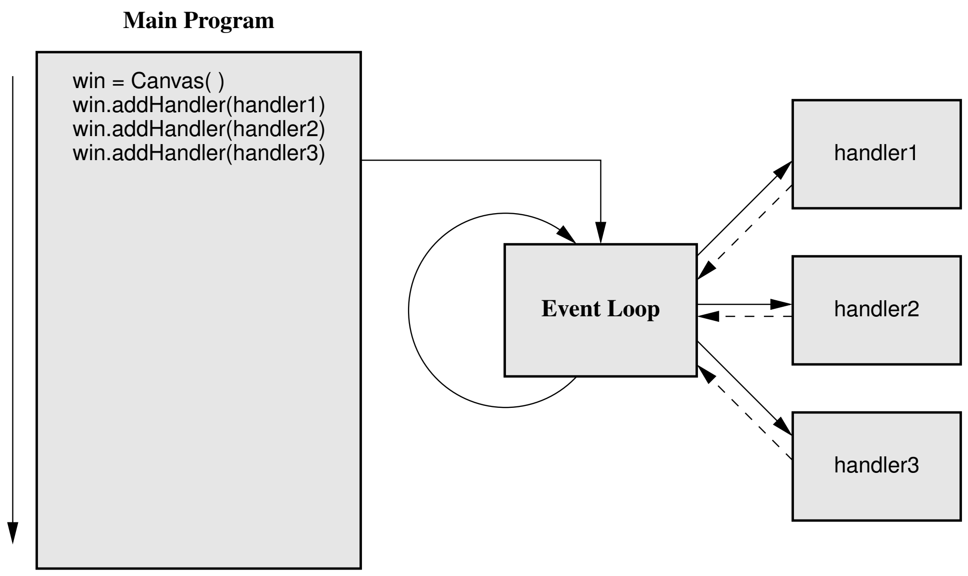

**FIGURE 15.1:** The flow of control in an event-driven program.

## 15.2 Event Handling in Our Graphics Module

The precise model for event-driven programming depends not only on the programming language, but on the underlying package that is being used for the graphical user interface. The graphics package receives the low-level user events, such as mouse clicks, and must decide how to process them. For the remainder of this chapter, we describe event handling with the cs1graphics module. As described in Chapter 3, that package supports a simple wait( ) model for basic support. However, it also supports more robust event handling.

### 15.2.1 The Event Loop

Without knowing it, every time you have used the cs1graphics package, an event loop has been running concurrently with the rest of your program. Every time you have clicked on the canvas's window or typed on the keyboard the event loop was informed. In most cases nothing occurred, as no handlers had been associated with the events. One exception is when you clicked on the icon to close the window; the event loop triggered a built-in handler that closed the window. When all canvas windows were closed, the event loop terminated. This starting and stopping of the event loop happens automatically.

### 15.2.2 The EventHandler Class

The module includes a generic EventHandler class that should be used as a parent class when defining your own handlers. Figure 15.2 outlines the two methods of that class, namely the constructor and a handle method. The handle method of the EventHandler class does not itself do anything. This method is overridden by a child class to describe the proper action in case of an event. The parameter to the handle method is an instance

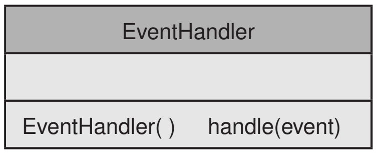

**FIGURE 15.2:** The EventHandler class.

of an Event class, used to describe the particular event that occurred. We will explore use of that parameter in Section 15.3. For now, we demonstrate a very simple handler which prints `Event Triggered` each time an event is detected:

```
class BasicHandler(EventHandler):
  def handle(self, event):
    print 'Event Triggered'
```

### Registering a handler with a graphics object

For an event handler to be active, it must be registered with one or more graphical objects. Both the Canvas and Drawable classes support two additional methods named addHandler and removeHandler to register and unregister a handler. For example, an instance of our BasicHandler class can be registered with a canvas as follows:

```
simple = BasicHandler( )
paper = Canvas( )
paper.addHandler(simple)
```

Each time the user clicks on the canvas window or presses a key while the canvas is active, an event is created triggering our handler which displays `Event Triggered`.

Using a similar technique, we can register an event handler directly with a particular drawable object. For example in the following scenario,

```
sun = Circle(30, Point(50,50))
sun.setFillColor('yellow')
paper = Canvas( )
paper.add(sun)
simple = BasicHandler( )
sun.addHandler(simple)          # register directly with the sun
```

the handler is only triggered when an event is received by the circle. Clicking on any other part of the canvas will not suffice. In our model, it is possible to register a single handler with multiple shapes or canvases, and to have multiple handlers registered with the same shape or canvas. If there are multiple handlers registered with an object and a relevant event is triggered, then *each* of those handlers is called in the order that they were registered. A single mouse click could initiate several reactions, one for each handler.

### 15.2.3 A Handler with State Information

In our first example, the handler did not have any state information. As our next example, we create a handler that counts the number of times that it has been triggered.

```
class CountingHandler(EventHandler):
  def __init__(self):
    EventHandler.__init__(self)   # call the parent constructor!
    self._count = 0

  def handle(self, event):
    self._count += 1
    print 'Event Triggered. Count:', self._count
```

In this example, we define our own constructor to establish the initial state of the _count attribute, which is later used from within the handle routine. Note that when overriding the constructor, we call the parent constructor so that the underlying state for the handler is properly established. In contrast, there is no need to call the parent version of the handle method, because that method is clearly documented as one that does not do anything.

As a further example, Figure 15.3 provides a program that updates a *graphical* count, in the form of a Text object, each time the user clicks on the canvas. In this case, the handler must be given a reference to the existing Text object in order to manipulate it. Therefore, we send this text object as a parameter when instantiating the actual handler at line 15. The reference to this object is then stored as an instance variable within the handler, at line 5, so that it can be used from within handle at line 10.

```
 1  class TallyHandler(EventHandler):
 2    def __init__(self, textObj):
 3      EventHandler.__init__(self)
 4      self._count = 0
 5      self._text = textObj
 6      self._text.setMessage(str(self._count))   # reset to 0
 7
 8    def handle(self, event):
 9      self._count += 1
10      self._text.setMessage(str(self._count))
11
12  paper = Canvas(100, 100)
13  score = Text('', 12, Point(40,40))
14  paper.add(score)
15  referee = TallyHandler(score)         # create the handler
16  paper.addHandler(referee)             # activate the handler
```

**FIGURE 15.3:** A handler that increments a graphical counter.

```
class HandleOnce(EventHandler):
  def __init__(self, eventTrigger):
    EventHandler.__init__(self)
    self._trigger = eventTrigger

  def handle(self, event):
    print "That's all folks!!!"
    self._trigger.removeHandler(self)

paper = Canvas( )
oneTime = HandleOnce(paper)
paper.addHandler(oneTime)
```

**FIGURE 15.4:** An event handler that can only be triggered once.

By default, handlers continue responding to events as long as the program runs. In contrast, we may want to have a handler that intentionally unregisters itself the first time it is triggered. This provides a one-time handling mechanism that is more naturally accomplished from within the context of the handler class. Figure 15.4 shows an example of this. When the handler is triggered, the call to removeHandler deactivates itself. So the first click on the canvas will trigger the print statement, but subsequent clicks will not.

## 15.3 The Event Class

Sometimes, an event handler may need information about the triggering event, such as the mouse location or the type of event that was received. To support this, the handle method of our EventHandler class is always passed an additional parameter that is an instance of an Event class, as originally introduced in Section 3.8.

Thus far in this chapter, you will see that all of our handlers have a signature

```
  def handle(self, event):
```

although they do not use the event parameter within the body. To use that parameter, we need to know more about the Event class. One method that it supports is getTrigger( ), which returns a reference to the underlying object upon which the event was originally triggered (a canvas or drawable object). This knowledge can often be used to simplify the design of a handler. For example, the HandleOnce class defined originally in Figure 15.4 needed to call the removeHandler method upon that trigger object. In that implementation we passed the identity of that trigger as a parameter to the constructor so that the handler could access it from within the handle method. However, we could have written the code more succinctly using getTrigger, as shown in Figure 15.5.

The Event class also supports a getDescription( ) accessor that returns a string indicating the kind of event that occurred. In the remainder of this section, we discuss several kinds of events, most notably those involving mouse or keyboard activity.

```
class HandleOnce(EventHandler):
  def handle(self, event):
    print "That's all folks!!!"
    event.getTrigger( ).removeHandler(self)

paper = Canvas( )
oneTime = HandleOnce( )
paper.addHandler(oneTime)
```

**FIGURE 15.5:** Another implementation of an event handler that can only be triggered once.

### 15.3.1 Mouse Events

There are several distinct kinds of mouse events: the mouse might be single clicked, released, or dragged while the button is being held down. The particular kind of event can be determined by calling getDescription( ), which returns a string, `'mouse click'`, `'mouse release'`, or `'mouse drag'` respectively. If the user clicks the button and then releases it two events will be triggered, the first identified as `mouse click` and the second as `mouse release`. When the mouse is dragged across the screen the sequence of events sent to the handler is a mouse click, followed by one or more mouse drags, and finally a mouse release event.

For some applications, we might be interested in the precise position of the mouse at the time the event occurred. The getMouseLocation( ) method of the Event class returns a Point instance, identifying the location of the mouse with respect to the coordinate system of the object that triggered the event.

As a first example using these techniques, Figure 15.6 gives a program that places circles on a canvas centered where the mouse is clicked. The handle function checks, at line 3, whether the event corresponded to a mouse click. If so, then line 4 create a new circle centered at the mouse location, and line 5 adds this circle to the underlying canvas. Notice that this handler intentionally ignores other kinds of events, such as when the mouse

```
1 class CircleDrawHandler(EventHandler):
2   def handle(self, event):
3     if event.getDescription( ) == 'mouse click':
4       c = Circle(5, event.getMouseLocation( ))
5       event.getTrigger( ).add(c)
6
7 paper = Canvas(100, 100)
8 handler = CircleDrawHandler( )
9 paper.addHandler(handler)
```

**FIGURE 15.6:** A program for drawing circles on a canvas.

is released or a key is pressed. Observe that the circle is created when the mouse button is pressed down and if the mouse is dragged it creates a circle when the button is first depressed. Try changing the code so that it draws the circle on a `mouse release` and observe the difference.

In the case of a `mouse drag` event, the getOldMouseLocation( ) method indicates where the mouse was before the dragging occurred. The combination of this knowledge and that of the current location can be used to figure out how far the mouse was moved. We will demonstrate use of this on a later example, in Section 15.4, showing how to drag objects across the screen.

Dragging the mouse across the screen triggers a `mouse click` event, followed by one or more `mouse drag` events, and finally a `mouse release` event. In some cases, we want a program to respond differently to a mouse drag than to a stationary click and release. This requires a handler that keeps track of the state as shown in Figure 15.7.

The attribute _mouseDragged is used to keep track of whether we are in the middle of a sequence of events corresponding to a mouse drag. The handle method in lines 6–15 responds to each mouse event. Note that the actual response is triggered when the mouse is released and other events are used to update the object’s state. When the mouse is first clicked on _mouseDragged is set to **False**. This ensures that if the next event is a `mouse release` then the handler will indicate that the mouse was clicked without dragging. However, if a `mouse drag` event occurs before the mouse button is released then as soon as the mouse button is released we are informed that the mouse was dragged.

```
 1  class ClickAndReleaseHandler(EventHandler):
 2    def __init__(self):
 3      EventHandler.__init__(self)
 4      self._mouseDragged = False
 5
 6    def handle(self, event):
 7      if event.getDescription( ) == 'mouse click':
 8        self._mouseDragged = False
 9      elif event.getDescription( ) == 'mouse drag':
10        self._mouseDragged = True
11      elif event.getDescription( ) == 'mouse release':
12        if self._mouseDragged:
13          print 'Mouse was dragged'
14        else:
15          print 'Mouse was clicked without dragging'
16
17  paper = Canvas( )
18  dragDetector = ClickAndReleaseHandler( )
19  paper.addHandler(dragDetector)
```

**FIGURE 15.7:** Differentiating between a mouse click and release and mouse dragging.

```
 1  class KeyHandler(EventHandler):
 2    def __init__(self, textObj):
 3      EventHandler.__init__(self)
 4      self._textObj = textObj
 5
 6    def handle(self, event):
 7      if event.getDescription( ) == 'keyboard':
 8        self._textObj.setMessage(self._textObj.getMessage( ) + event.getKey( ))
 9      elif event.getDescription( ) == 'mouse click':
10        self._textObj.setMessage('')  # clear the text
11
12  paper = Canvas( )
13  textDisplay = Text('', 12, Point(100,100))    # empty string initially
14  paper.add(textDisplay)
15  echo = KeyHandler(textDisplay)
16  paper.addHandler(echo)
```

**FIGURE 15.8:** A program for echoing characters upon a canvas.

### 15.3.2 Keyboard Events

When the user presses a key on the keyboard, this triggers a keyboard event upon whatever object currently has the “focus.” Depending upon the operating system, the focus is determined either by where the mouse is currently located or perhaps which object was most recently clicked. From within a handler, this type of event is reported as `'keyboard'` by getDescription( ). If needed, the getMouseLocation( ) is supported for a keyboard event. But keyboard events also support a behavior getKey( ) that returns the single character that was typed on the keyboard to trigger the event. Note that if the user types a series of characters, each one of those triggers a separate event.

Figure 15.8 gives a simple program showing how to display characters graphically as they are typed within a canvas. The main flow begins by creating a canvas and adding a new Text instance, although one that displays the empty string. When the handler is instantiated at line 15, we must explicitly send it a reference to this underlying Text object.

Within the class definition, we record the identity of the text object as an attribute at line 4 and then use it within the handle body at lines 8 and 10. When a user types a character, that character is added to the displayed string. If the user clicks on the canvas, the text is reset to the empty string.

### 15.3.3 Timers

Mouse and keyboard events are originally triggered as a direct result of the user’s activities. When using event-driven programming, there are other scenarios in which we want our program to respond to an event that we generate internally. The cs1graphics module includes a definition of a Timer class for this purpose. A timer is not itself a graphical object, nor is it an event or an event handler. A timer instance is a self-standing object that is used to *generate* new events (almost like a ghost who generates events as if a user).

A Timer has an interval, measured in seconds, that can be specified as a parameter to the constructor. Thus the syntax Timer(10) creates a timer that generates an event after ten seconds has elapsed. However, the timer does not start counting upon instantiation; the start( ) method must be called to begin the timer's clock. After the specified time interval passes, it generates a timer event. The constructor for the Timer class takes an optional second parameter that can be used to specify if the timer should be automatically restarted after it triggers an event. By default, the timer does not restart itself. However, a timer constructed with the call Timer(5, **True**) will repeatedly trigger an event every five seconds, once it is started. Its internal timer can be explicitly interrupted with the stop( ) method.

For a timer to be useful, there must be a corresponding handler that is registered to listen for those events. For example, we could register one of our CountingHandler instances from page 498 to listen to a timer, creating the following event-driven stopwatch:

```
alarm = Timer(1, True)
stopwatch = CountingHandler( )
alarm.addHandler(stopwatch)
print 'Ready...'
alarm.start( )                      # yet never stops...
```

Once per second, an event is generated, triggering the handler. The user will see the count displayed each second. Notice that the earlier CounterHandler method handles all events in the same way, and therefore we do not need to do anything special for it to recognize the `timer` event. We could similarly use such a timer to trigger some action on the canvas. For example, animating a rotating shape.

```
class RotationHandler(EventHandler):
  def __init__(self, shape):
    self._shape = shape

  def handle(self, event):
    self._shape.rotate(1)

paper = Canvas(100,100)
sampleCircle = Circle(20, Point(50,20))
sampleCircle.adjustReference(0,30)
paper.add(sampleCircle)

alarm = Timer(0.1, True)
rotator = RotationHandler(sampleCircle)
alarm.addHandler(rotator)
alarm.start( )
```

This creates a circle and draws it on the canvas and every tenth of a second rotates around the reference point (the center of the canvas). The circle will continue on its path around the screen until the canvas is closed.

### 15.3.4 Monitors

In some cases, we want to blend event-driven programming with the more sequential flow-driven approach. For example, we have already seen how a command like **raw_input** causes the current flow of control to wait indefinitely until the user responds. Sometimes we want a similar way to wait indefinitely, until triggered by some combination of events.

For simple cases, we provided the wait( ) behavior as introduced in Chapter 3 and reviewed on page 494 of this chapter. However, that behavior cannot be customized. It blocks the flow of control until any kind of event is triggered upon the one particular canvas or drawable object. There are times where we may want to wait until one particular kind of event occurs, or to wait until an event happens on any one of a larger set of objects.

To this end, we introduce a Monitor class. The class is so named because it can be thought of as “monitoring” some condition and alerting us once that condition is met. The class supports two methods, wait( ) and release( ). When wait( ) is called, control of that flow will not be returned until the monitor is somehow released, presumably by some event handler. Yet we can use many different handlers to potentially release a given monitor.

Figure 15.9 gives an example of the use of a monitor. We create a Monitor instance at line 11, and then a dedicated handler at line 12, which will later be used to release the

```
 1  class ShapeHandler(EventHandler):
 2    def __init__(self, monitor):
 3      EventHandler.__init__(self)
 4      self._monitor = monitor
 5
 6    def handle(self, event):
 7      if event.getDescription( ) == 'mouse drag':
 8        self._monitor.release( )
 9
10  paper = Canvas( )
11  checkpoint = Monitor( )
12  handler = ShapeHandler(checkpoint)
13
14  cir = Circle(10, Point(50,50))
15  cir.setFillColor('blue')
16  cir.addHandler(handler)
17  paper.add(cir)
18  square = Square(20, Point(25,75))
19  square.setFillColor('red')
20  square.addHandler(handler)
21  paper.add(square)
22
23  checkpoint.wait( )
24  paper.setBackgroundColor('green')
```

**FIGURE 15.9:** The use of a monitor.

monitor. Notice that the handler is registered with both the circle and the square, and that it is programmed to only respond to mouse drag events. At line 23, the wait( ) method of the monitor is invoked, essentially blocking the main flow of control. Line 24 is not immediately executed. In fact, it will not be executed until the monitor is released. This will only occur when the user drags the mouse on either the circle or square; at that point the original flow of control continues to line 24 and the background color is changed. Technically, the handler continues listening for mouse drags, rereleasing the monitor each time. However a rerelease has no subsequent effect. Notice that the use of a monitor has a similar style to invoking wait( ) directly upon a drawable object, except that it allows us to wait for only specific kinds of events, or to simultaneously wait upon multiple objects.

## 15.4 Programming Using Events

Our preliminary examples have been rather simple, demonstrating each new concept individually. In this section, we combine the basic techniques to produce three programs with more interesting results.

### 15.4.1 Adding and Moving Shapes on a Canvas

As our next example, we design a program to create and modify shapes. Each time the mouse is clicked on the background, a new shape is created and added to the canvas. Dragging the mouse on an existing shape can be used to move it, while clicking on it enlarges the shape. Typing on the keyboard when a shape has the focus changes its color.

We use two different handlers for this task, one that listens for events on the background of the canvas, and a second that listens for events on any of the existing shapes. The complete program is given in Figure 15.10. Let’s examine how that program works. At lines 1–19, we define a ShapeHandler class. This will eventually be used to manage the shapes that have already been added to our canvas. The handler has three distinct behaviors, depending upon whether the user is dragging the mouse, clicking on the shape, or typing on the keyboard. The handler needs to respond differently to mouse drags than to clicking and releasing the mouse. This will be dealt with using the same style as the program in Figure 15.7. When dragging, notice that we use the difference between the new mouse location and the old mouse location to determine how far to move the shape. When the user triggers a keyboard event, our current implementation switches to a random color, although we could easily modify this code to pick a specific color depending upon what letter was typed (see Exercise 15.4).

In lines 21–42, we define a NewShapeHandler that will listen for events on the canvas background. Each time the user single clicks, a new shape is added to the canvas at that location. This implementation automatically cycles through four different shapes. Within the constructor, a _shapeCode attribute is initialized to control which type of shape is next created (that integer code is later “advanced” at line 37). The other attribute created in the constructor, at line 25, is an instance of the ShapeHandler class. We will use a single handler to manage all of the shapes that are created. This handler is registered to each new shape at line 42.

With these class definitions, the main part of the program creates a canvas, and then instantiates a NewShapeHandler while immediately registering it with the canvas. Once this is done, the user may begin playing.

```python
class ShapeHandler(EventHandler):
  def __init__(self):
    EventHandler.__init__(self)
    self._mouseDragged = False

  def handle(self, event):
    shape = event.getTrigger( )
    if event.getDescription( ) == 'mouse drag':
      old = event.getOldMouseLocation( )
      new = event.getMouseLocation( )
      shape.move(new.getX( )−old.getX( ), new.getY( )−old.getY( ))
      self._mouseDragged = True
    elif event.getDescription( ) == 'mouse click':
      self._mouseDragged = False
    elif event.getDescription( ) == 'mouse release':
      if not self._mouseDragged:
        shape.scale(1.5)
    elif event.getDescription( ) == 'keyboard':
      shape.setFillColor(Color.randomColor( ))

class NewShapeHandler(EventHandler):
  def __init__(self):
    EventHandler.__init__(self)
    self._shapeCode = 0
    self._handler = ShapeHandler( )  # single instance handles all shapes

  def handle(self, event):
    if event.getDescription( ) == 'mouse click':
      if self._shapeCode == 0:
        s = Circle(10)
      elif self._shapeCode == 1:
        s = Square(10)
      elif self._shapeCode == 2:
        s = Rectangle(15,5)
      elif self._shapeCode == 3:
        s = Polygon(Point(5,5), Point(0,−5), Point(−5,5))
      self._shapeCode = (self._shapeCode + 1) % 4  # advance cyclically

      s.move(event.getMouseLocation( ).getX( ), event.getMouseLocation( ).getY( ))
      s.setFillColor('white')
      event.getTrigger( ).add(s)        # add shape to the underlying canvas
      s.addHandler(self._handler)       # register the ShapeHandler with the new shape

paper = Canvas(400, 300, 'white', 'Click me!')
paper.addHandler(NewShapeHandler( ))  # instantiate handler and register all at once
```

**FIGURE 15.10:** Creating and modifying shapes using events.

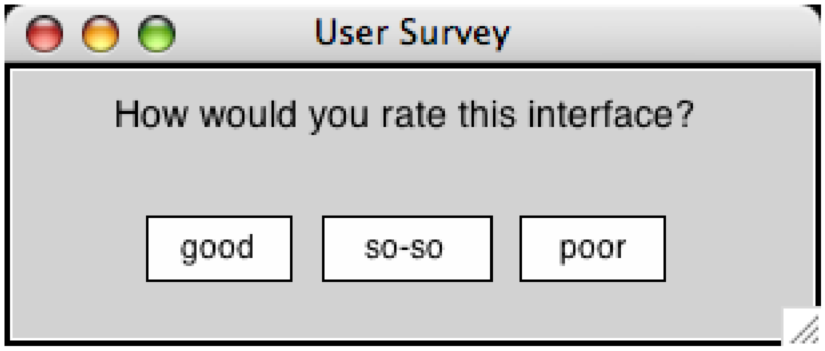

**FIGURE 15.11:** A screenshot of a sample dialog box.

### 15.4.2 A Dialog Box Class

Our second advanced example is to create what is known as a dialog box. This is a common technique for blending the event-driven and flow-driven styles. Often graphics-based programs need to ask a specific question of the user and to await that response before proceeding. For example when closing an open file, a pop-up window might appear warning the user that all data will be lost and asking if they want to continue. Buttons may offer alternatives such as "OK" or "Cancel" that the user can click on to respond. Other dialogs may offer choices like "Yes" and "No", or more than two choices such as "Save", "Exit without saving", and "Cancel". A typical sample of a dialog box is shown in Figure 15.11.

To support such dialogs, we will design a utility class Dialog. Before looking at the implementation, we focus on the programming interface for those wanting to use our dialogs. The constructor is called with the signature

```
Dialog(prompt, options, title, width, height)
```

although our implementation will offer default values for these parameters. The prompt is a string that will be displayed as part of the dialog box, options is a sequence of strings to be offered as choices, and the parameters title, width, and height affect the window settings.

Instantiating a Dialog instance configures it, but does not display it. The user interaction is accomplished by calling the display( ) method, which waits for a user response and then returns the string corresponding to the chosen option. The dialog can be redisplayed whenever the same interaction is needed. An example of its use might appear as follows:

```
survey = Dialog('How would you rate this interface?',
                ('good', 'so-so', 'poor'), 'User Survey')
answer = survey.display( )      # waits for user response
if answer != 'good':
  print "Let's see you do better (see exercises)"
```

When displayed, the user would see the window shown in Figure 15.11. Once the user clicks on a button, the answer is returned and our program continues.

```
class Dialog(EventHandler):                              # Note: handles itself!
  """Provides a pop-up dialog box offering a set of choices."""

  def __init__(self, prompt='Continue?', options=('Yes', 'No'),
               title = 'User response needed', width=300, height=100):
    """Create a new Dialog instance but does not yet display it.

    prompt    the displayed string (default 'Continue?')
    options   a sequence of strings, offered as options (default ('Yes', 'No') )
    title     string used for window title bar (default 'User response needed')
    width     width of the pop-up window (default 300)
    height    height of the pop-up window (default 100)
    """
    EventHandler.__init__(self)
    self._popup = Canvas(width, height, 'lightgray', title)
    self._popup.close( )                             # hide, for now
    self._popup.add(Text(prompt, 14, Point(width/2,20)))

    xCoord = (width − 70*(len(options)−1))/2         # Center buttons
    for opt in options:
      b = Button(opt, Point(xCoord, height−30))
      b.addHandler(self)                             # we will handle this button ourselves
      self._popup.add(b)
      xCoord += 70

    self._monitor = Monitor( )
    self._response = None

  def display(self):
    """Display the dialog, wait for a response and return the answer."""
    self._response = None                            # clear old responses
    self._popup.open( )                              # make dialog visible
    self._monitor.wait( )                            # wait until some button is pressed
    self._popup.close( )                             # then close the popup window
    return self._response                            # and return the user's response

  def handle(self, event):
    """Check if the event was a mouse click and have the dialog return."""
    if event.getDescription( ) == 'mouse click':
      self._response = event.getTrigger( ).getMessage( )   # label of chosen option
      self._monitor.release( )                              # ready to end dialog
```

**FIGURE 15.12:** A dialog box class.

The complete implementation of the Dialog class is given in Figure 15.12. The constructor initializing a pop-up window, but immediately hides the window at line 16 for the time being. This allows us to work behind the scene in configuring our dialog box based upon the given parameters. We add a text prompt and, below that, a row of buttons labeled with the indicated options. Our initial implementation does a rough job of guessing at the geometry (designing a more robust layout is left as Exercise 15.5).

Since the constructor is responsible for preparing this dialog for future use, it also creates a Monitor instance at line 26 and initializes a response to **None** at line 27. The monitor will be used to intentionally delay the progression of code until the user has chosen one of the buttons. That user interaction is started by calling the display method. This method resets the response to empty, makes the pop-up window visible, and then waits on the monitor at line 33. We are not yet prepared to give an answer to the caller of this method. However, once a button has been pressed, the monitor will be released and we go on to lines 34 and 35, to close the dialog window and to return the chosen response.

The actual code for handling the buttons is given in lines 37–41. There is a great deal of subtlety tying all of this code together. Most significantly, we use inheritance at line 1 of our class definition to declare the Dialog instance itself as an EventHandler. We could have used two different classes, one for the dialog window and the other for a handler class, but the sharing of information is much more convenient as a single object. Notice that the event handling code needs to access two objects, as it must store the chosen response in a way that can later be accessed by the dialog code, and it must release the dialog's monitor so that the original call can proceed. If we were to create a separate handler, we would have to construct that handler while passing it references to those underlying objects (as was done with ShapeHandler in Figure 15.9). By having our class serve as its own handler, it already has the needed access! When a button is pressed, the chosen option string is retrieved at line 40 from the specific button that triggered the current event, and then the monitor is released at line 41 to allow the primary flow of control within the display method to continue beyond the wait initiated at line 33.

### 15.4.3 A Stopwatch Widget

As our third example, we design a basic stopwatch. Unlike the Dialog class, which brings up its own pop-up window, our stopwatch is designed to be a component that can itself be used on a canvas. Such graphical components are often called ***widgets***. A picture of our stopwatch is shown in Figure 15.13. It displays a time, measured in minutes and seconds, and has three buttons. The left button is used to start (or restart) the stopwatch. The middle button is used to stop the watch, and the third button is used to reset the time to zero.

Again, we begin by considering the *use* of our presumed class. We offer the following simple demonstration.

```
paper = Canvas(400,400)
clock = Stopwatch( )
paper.add(clock)
clock.move(200,200)
```

Notice that the Stopwatch instance is added to the canvas as with any Drawable object.

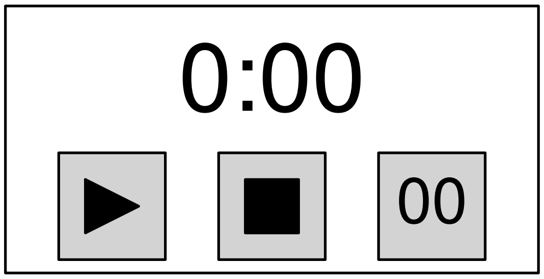

**FIGURE 15.13:** A screenshot of the Stopwatch widget.

Our Stopwatch design is based upon the use of multiple inheritance. We want a stopwatch to be treated as a single graphical component, and so we inherit from the Layer class (as done in Section 9.4.3). At the same time, we find it convenient to have our objects handle their own events, as we did with the preceding Dialog example, so our class inherits from EventHandler as well. Therefore, we use multiple inheritance and begin the class definition with the syntax **class** Stopwatch(Layer, EventHandler). We will ensure that our constructor calls the parent form of the constructor for both the Layer and EventHandler classes.

The complete implementation is given in Figure 15.14. Much of the constructor is dedicated to laying out the graphical components of the stopwatch, namely a border, a text display, three buttons, and appropriate icons. The display, the buttons and the icons are designated as data members so that we can reference them from the event handling routine. The border is identified with a local variable because we will not need to access it when handling an event. The loop at lines 28 and 29 is used to explicitly add all of our intended components to ourself (as a Layer).

Perhaps the most interesting part of the constructor is given in lines 31–34. Here we initialize our internal clock to zero and create a periodic Timer object (although one that is not yet started). In lines 33 and 34 we formally register our Stopwatch instance as the handler for the event-triggering components.

The getTime method at lines 36–42 is a simple utility that converts the current internal clock to an appropriately formatted string for displaying the time in minutes and seconds (we do not use hours as a unit of time, although this could easily be accomplished). We see this utility used from lines 51 and 59 of the subsequent handle routine.

The complete handle routine is given in lines 43–59. It is written to handle potential events from four different components. If an event is triggered by the timer, we update our internal clock and refresh the displayed time (lines 49–51). If the event is a mouse click, it must have occurred on one of the three buttons; our response depends upon which button. When the start button is pressed, we start the internal timer (even if it was already running), and similarly we stop the timer when the user clicks on the stop button. Otherwise, the reset button was pressed, so we reset the internal clock to zero and update the displayed time. Notice that our reset behavior does not change the Timer instance. Pressing reset when the clock is running causes it to go back to zero but it continues to run. Pressing the reset button on a stopped clock, resets it to zero while leaving the clock stopped.

```
class Stopwatch(Layer, EventHandler):
  """Display a stopwatch with start, stop, and reset buttons."""

  def __init__(self):
    """Create a new Stopwatch instance."""
    Layer.__init__(self)
    EventHandler.__init__(self)

    border = Rectangle(200,100)
    border.setFillColor('white')
    border.setDepth(52)
    self._display = Text('0:00', 36, Point(0,−20))

    self._start = Square(40, Point(−60,25))
    self._stop  = Square(40, Point(0,25))
    self._reset = Square(40, Point(60,25))
    buttons = [self._start, self._stop, self._reset]
    for b in buttons:
      b.setFillColor('lightgray')
      b.setDepth(51)               # in front of border, but behind icons

    self._startIcon = Polygon( Point(−70,15), Point(−70,35), Point(−50,25) )
    self._startIcon.setFillColor('black')
    self._stopIcon = Square(20, Point(0,25))
    self._stopIcon.setFillColor('black')
    self._resetIcon = Text('00', 24, Point(60,25))
    buttons.extend([self._startIcon, self._stopIcon, self._resetIcon])
    for obj in buttons + [self._display, border]:
      self.add(obj)                # add to the layer

    self._clock = 0                # measured in seconds
    self._timer = Timer(1, True)
    for active in [self._timer] + buttons:
      active.addHandler(self)      # we will handle all such events

  def getTime(self):
    """Convert the clock's time to a string with minutes and seconds."""
    min = str(self._clock // 60)
    sec = str(self._clock % 60)
    if len(sec) == 1:
      sec = '0'+sec                # pad with leading zero
    return min + ':' + sec
```

**FIGURE 15.14:** Implementation of a Stopwatch class (continued on next page).

```
  def handle(self, event):
    """Deal with each of the possible events.

    The possibilities are timer events for advancing the clock,
    and mouse clicks on one of the buttons.
    """
    if event.getDescription( ) == 'timer':
      self._clock += 1
      self._display.setText(self.getTime( ))
    elif event.getDescription( ) == 'mouse click':
      if event.getTrigger( ) in (self._start, self._startIcon):
        self._timer.start( )
      elif event.getTrigger( ) in (self._stop, self._stopIcon):
        self._timer.stop( )
      else: # must have been self._reset or self._resetIcon
        self._clock = 0
        self._display.setText(self.getTime( ))

if __name__ == '__main__':
  paper = Canvas(400,400)
  clock = Stopwatch( )
  paper.add(clock)
  clock.move(200,200)
```

**FIGURE 15.14 (continuation):** Implementation of a Stopwatch class.

## 15.5 Case Study: a Full GUI for Mastermind

If we combine all of these techniques for dealing with events we can write a full graphical interface for the Mastermind game developed in Chapter 7. Our goal is to have the user control the entire game, from start to finish, with the mouse. Figure 15.15 shows a screenshot of the game in action; to select a peg color, the user clicks on the peg and our interface displays a box of choices. Once all of the peg colors have been selected the button labeled “Guess” is activated; pressing this button finalizes the guess. This provides a fairly natural user interface with minimal changes made to the original code.

For flexibility, the design of the Mastermind program had separate classes for dealing with user input and output. For input we used the TextInput class and for output we used either the TextOutput or GraphicsOutput classes. A full GUI for the game involves both input and output involving the same graphical objects, so a single class MastermindGUI will be written to take the place of both the input and output classes. See Figure 15.16 for the methods it must support.

Since all of the display functions are the same as in GraphicsOutput, we will inherit the GUI class from it. However, all of the input routines need to be completely rewritten since they were originally text based. Below are the changes to the main routine for starting the Mastermind game. First an instance of the MastermindGUI is created. Since the new class handles both input and output there is no need for two separate class instances being

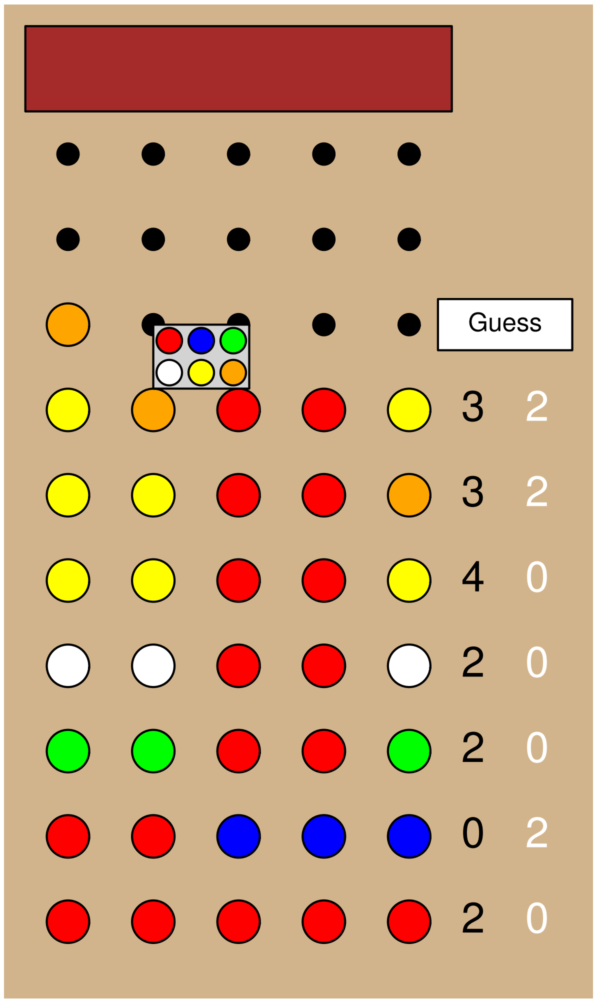

**FIGURE 15.15:** A screenshot for the new interface to Mastermind.

sent to the constructor for Mastermind. These are the only changes to necessary when starting the game.

```
if __name__ == '__main__':
  palette = ('Red', 'Blue', 'Green', 'White', 'Yellow', 'Orange',
             'Purple', 'Turquoise')
  interface = MastermindGUI(palette)
  game = Mastermind(interface, interface)
```

The full implementation of MastermindGUI is in Figure 15.17. The first group of methods are used to query the user for game parameters in lines 9–29. Comparing the code to the TextInput class on pages 259–260 of Chapter 7, we see that each call to the

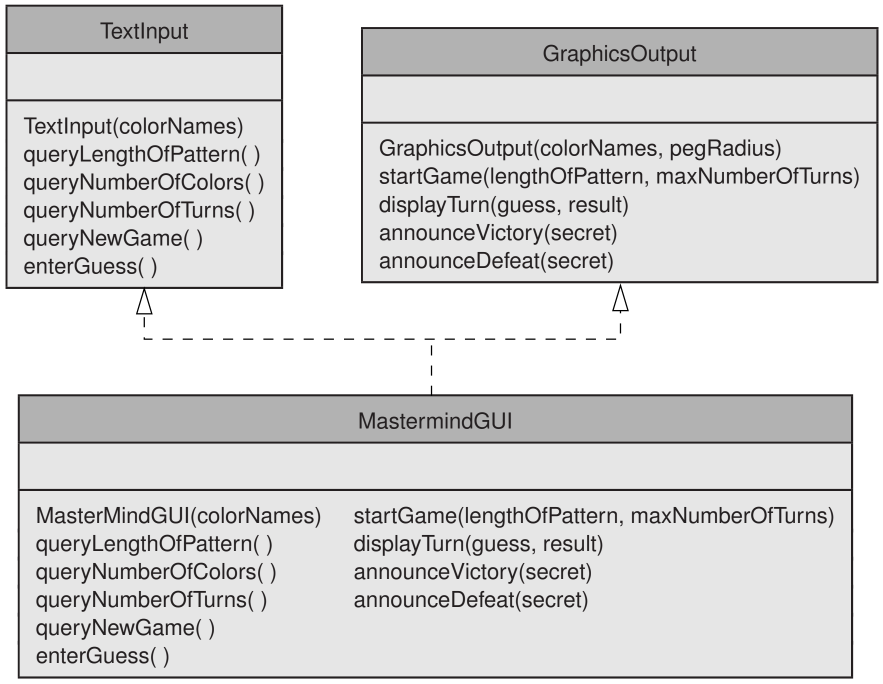

**FIGURE 15.16:** The MastermindGUI must perform all of the tasks of both the input and output classes.

_readInt function has been replaced by the creation of a dialog box. This utilizes the Dialog class written in Section 15.4 to display the possible options for each query. This provides a straightforward way to use existing code to enter these game parameters. (In Exercise 15.9 you will have an opportunity to improve upon this user interface.)

The GraphicsOutput class has a method _setupBackground that is used to render the game board. This method is overridden in lines 36–51. When it comes time for the user to enter the pattern we will need to add handlers for the pegs so that the user can click on them to select a color. To do this we need to keep track of each peg, so we store the location of the Circle instances for each peg location in the member variable _holes.

The enterGuess method in lines 53–78 allows the user to enter a pattern using the mouse. When the user clicks on a peg, a small rectangle appears with the color options. When the user selects a color, a peg of that color peg is placed in the hole. The user can change the color by selecting that peg location again. When the user is done, he presses the button labeled “Guess.” For this to operate properly, we need two handler classes: PegHandler for clicking on the peg, and ButtonHandler for clicking the button. When enterGuess is called, a PegHandler instance is created for each hole in the current row of the board and associated to the corresponding circle in lines 59–61. In lines 63–67 a Button and associated handler are created to respond when the user wishes to finalize the guess. Much of the complexity of the two handler implementations is present to ensure that a color must be selected for each peg before the “Guess” button is pressed. The attribute _pegEntered keeps track of which pegs have had their colors selected.

The PegHandler class is implemented in lines 79–122. The constructor initializes an attribute to keep track of the MastermindGUI instance, another to track which peg is being selected, and a monitor for later use. When the peg is clicked on, the handle method renders a rectangle containing a circle for each color option. Each circle has an instance of ChoiceHandler that is triggered when that color is selected. We wait on the monitor at line 119, until it is released by line 140 of the ChoiceHandler.handle method. That handler also sets the color choice for that peg and indicates that the peg color has been selected.

```
 1  from cs1graphics import *
 2  from Mastermind import Mastermind
 3  from GraphicsOutput import GraphicsOutput
 4  from Dialog import Dialog
 5  from Pattern import Pattern
 6
 7  class MastermindGUI(GraphicsOutput):
 8    """Class to provide full graphical interface to Mastermind"""
 9    def queryLengthOfPattern(self):
10      """Ask the user how many pegs in the secret pattern."""
11      dialog = Dialog('How many pegs are in the secret?',
12                      ['4', '6', '8', '10'], 'Length of pattern')
13      return int(dialog.display( ))
14
15    def queryNumberOfColors(self):
16      """Ask the user how many colors to use for secret pattern."""
17      inputOptions = [ ]
18      for i in range(2,len(self._colorNames)+1):
19        inputOptions.append(str(i))
20      dialog = Dialog('How many colors are available?',
21                      inputOptions, 'Number of color', 500)
22      self._numberOfColors = int(dialog.display( ))
23      return self._numberOfColors
24
25    def queryNumberOfTurns(self):
26      """Ask the user maximum number of guesses to be allowed."""
27      dialog = Dialog('How many turns are allowed?',
28                      ['5', '10', '15', '20'], 'Number of turns')
29      return int(dialog.display( ))
30
31    def queryNewGame(self):
32      """Offer the user a new game. Return True if accepted, False otherwise."""
33      dialog = Dialog('Would you like to play again?',
34                      ['Yes', 'No'], 'Again?')
35      return dialog.display( ) == 'Yes'
```

**FIGURE 15.17:** The GUI for Mastermind (continued on next page).

```
  def _setupBackground(self):
    """Draws the backgound to the graphics canvas."""
    block = Rectangle(4*self._lengthOfPattern*self._pegRadius, 4*self._pegRadius)
    block.move((1 + 2*self._lengthOfPattern)*self._pegRadius, 3*self._pegRadius)
    block.setFillColor('brown')
    block.setDepth(60)
    self._canvas.add(block)

    self._holes = [ ]
    for row in range(self._maxNumberOfTurns):
      self._holes.append( [None] * self._lengthOfPattern )
      for col in range(self._lengthOfPattern):
        self._holes[row][col] = Circle(self._pegRadius/2, self._getCenterPoint(row,col))
        self._holes[row][col].setFillColor('black')
        self._canvas.add(self._holes[row][col])
    self._canvas.refresh( )

  def enterGuess(self):
    """Have user enter guess and return response."""
    self._guess = Pattern(self._lengthOfPattern)

    handlers = [ ]                          # Turn on handlers for each of the pegs
    self._pegEntered = [False] * self._lengthOfPattern
    for i in range(self._lengthOfPattern):
      handlers.append(PegHandler(self, i))
      self._holes[self._currentTurnNum][i].addHandler(handlers[i])

    button = Button('Guess')                # Create a button with a handler
    button.move((4*self._lengthOfPattern+3.5)*self._pegRadius,
        (4*(self._maxNumberOfTurns−self._currentTurnNum−1)+7)*self._pegRadius)
    button.addHandler(ButtonHandler(self))
    self._canvas.add(button)
    self._canvas.refresh( )

    self._finalized = Monitor( )
    self._finalized.wait( )                 # Wait for guess to be finalized

    for i in range(self._lengthOfPattern):  # Remove the holes
      self._holes[self._currentTurnNum][i].removeHandler(handlers[i])
      self._canvas.remove(self._holes[self._currentTurnNum][i])
    self._canvas.remove(button)             # Remove the "guess" button

    return self._guess
```

**FIGURE 15.17 (continuation):** The GUI for Mastermind (continued on next page).

```python
class PegHandler(EventHandler):
  """Manager for an overlay to select a peg color."""
  def __init__(self, gui, peg):
    """Create a new PegHandler instance.

    gui   reference to the user interface
    peg  integer indicating which peg is being set
    """
    EventHandler.__init__(self)
    self._gui = gui
    self._peg = peg
    self._monitor = Monitor( )

  def handle(self, event):
    """Display options when the user mouse clicks on a peg."""
    if event.getDescription( ) == 'mouse click':
      hole = event.getTrigger( )
      hole.removeHandler(self)                # disable handler until we are done

      box = Layer( )                          # Create a box of options and display
      box.move(hole.getReferencePoint( ).getX( ), hole.getReferencePoint( ).getY( ))
      box.setDepth(0)
      self._gui._canvas.add(box)

      numColumns = (1 + self._gui._numberOfColors) // 2
      r = 0.6 * self._gui._pegRadius
      background = Rectangle(r*2.5*numColumns, r*5)
      background.move(background.getWidth( )/2, background.getHeight( ) / 2)
      background.setFillColor('light gray')
      background.setDepth(100)
      box.add(background)

      for i in range(2):
        for j in range(numColumns):
          if numColumns*i + j < self._gui._numberOfColors:
            peg = Circle(r, Point(r*(2.5*j+1.25), r*(2.5*i+1.25)))
            peg.setFillColor(self._gui._colorNames[numColumns*i + j])
            peg.addHandler(ChoiceHandler(self, numColumns*i + j))
            box.add(peg)
      self._gui._canvas.refresh( )          # Redisplay gameboard with the choices
      self._monitor.wait( )                 # Wait for a choice to be selected
      self._gui._canvas.remove(box)
      self._gui._canvas.refresh( )          # Redisplay gameboard without choices
      hole.addHandler(self)                 # Allow user to change her choice
```

**FIGURE 15.17 (continuation):** The GUI for Mastermind (continued on next page).

```
123 class ChoiceHandler(EventHandler):
124   """Handler class for the selection of a colored peg."""
125   def __init__(self, pegHandler, colorNum):
126     """Create a new instance."""
127     EventHandler.__init__(self)
128     self._pegHand = pegHandler
129     self._color = colorNum
130
131   def handle(self, event):
132     """Set the choice of color and close the popup."""
133     if event.getDescription( ) == 'mouse click':
134       gui = self._pegHand._gui
135       gui._guess.setPegColor(self._pegHand._peg, self._color)
136       hole = gui._holes[gui._currentTurnNum][self._pegHand._peg]
137       hole.setRadius(gui._pegRadius)
138       hole.setFillColor(gui._colorNames[self._color])
139       gui._pegEntered[self._pegHand._peg] = True
140       self._pegHand._monitor.release( )
141
142 class ButtonHandler(EventHandler):
143   """Handler for user submitting a guess."""
144   def __init__(self, gui):
145     """Create a new ButtonHandler."""
146     EventHandler.__init__(self)
147     self._gui = gui
148
149   def handle(self, event):
150     """Release the monitor waiting for a guess."""
151     if event.getDescription( ) == 'mouse click':
152       if False not in self._gui._pegEntered:          # all pegs have been chosen
153         self._gui._finalized.release( )
154
155 if __name__ == '__main__':
156   palette = ('Red', 'Blue', 'Green', 'White', 'Yellow', 'Orange',
157          'Purple', 'Turquoise')
158   interface = MastermindGUI(palette)
159   game = Mastermind(interface, interface)
```

**FIGURE 15.17 (continuation):** The GUI for Mastermind.

The ButtonHandler class in lines 142-153 responds each time the “Guess” button in pressed. However, the handler only finalizes the guess if every peg has been entered, as determined by examining the _pegEntered list stored in the MastermindGUI instance. When all of these values are **True** the monitor of the MastermindGUI class is released. This frees the flow of control that had been waiting at line 71 of the enterGuess method, thereby removing all of the handlers and pegs for the current row of the interface, and returning the finalized guess.

The code for entering the guess is rather complicated. But by carefully examining the classes and their interactions, it is possible to see how it works. Notice that it relies on several helper classes that need access to each others’ states in order to work together. With time and experience you will be able to write similar programs yourself.

## 15.6 Chapter Review

### 15.6.1 Key Points

- The flow of an event-driven program responds to user interaction. Each time an event occurs, code is run to respond appropriately.
- A routine to respond to events is called a handler. Every object that needs to respond to an event derives a class from EventHandler to process events for that object.
- An event loop waits for user interaction or other events to occur and call the appropriate handler. This loop will continue to run until the program is terminated.
- When an event occurs, the handle method of the handler class is called with an instance of Event as its only parameter. This instance stores the type of event that has occurred, the location of the mouse when the event was triggered and any other pertinent information.

### 15.6.2 Glossary

**callback function** *See* event handler.

**event** An external stimulus on a program, such as a user’s mouse click.

**event-driven programming** A style in which a program passively waits for external events to occur, responding appropriately to those events as needed.

**event handler** A piece of code, typically a function or class, used to define the action that should take place in response to a triggered event.

**event loop** A potentially infinite loop that does nothing other than wait for events. When an event is triggered, an event handler is called to respond appropriately. Once that event handler completes, the event loop is typically continued.

**listener** *See* event handler.

**thread** An abstraction representing one of possibly many active flows of control within an executing program.

**widget** An object that serves a particular purpose in a graphical user interface.

### 15.6.3 Exercises

**Practice 15.1:** Write a handler that will display the type of event and the mouse location whenever a mouse event has occurred.

**Practice 15.2:** Write a program that draws a circle in the center of a graphics window whenever the mouse button is clicked down, and erases it when the button is released.

**Exercise 15.3:** Write a program to draw circles on a canvas. When the canvas is first clicked a circle should appear. As the user drags the mouse the radius should change so that the mouse is on the boundary of the circle. Once the mouse button is released the circle's geometry should remain fixed.

**Exercise 15.4:** Our program for creating and modifying shapes, in Figure 15.10, picks an arbitrary color when it receives a keyboard event. Modify this program so that the user can select from a palette of colors based on the key that was typed (e.g., `'r'` for red, `'b'` for blue).

**Exercise 15.5:** The Dialog class from Figure 15.12 could be made more robust by having the geometry adjust properly to the number of options and the length of those strings. As originally implemented, the overall height and width of the pop-up window, and the spacing of the buttons is hardcoded. Rewrite that program to do a better job of laying out the geometry based upon the option list.

**Exercise 15.6:** Modify the Stopwatch of Section 15.4.3, so that it measures time in hundredths of a second, displayed as 0:00.00.

**Exercise 15.7:** Using techniques similar to the Stopwatch of Section 15.4.3, redesign the Mailbox from Section 9.6 so that the flag automatically toggles when it is clicked upon and so that the door opens and closes when it is clicked upon.

**Exercise 15.8:** Write a program to animate an analog clock face with hour, minute, and second hands. The program should take a time as input, and the clock should display this time and move the hands the appropriate amount each second.

**Exercise 15.9:** The Mastermind interface in the case study has separate dialog boxes to enter each game parameter. Modify the implementation so there is a single dialog box to enter all of the game parameters.

## Projects

**Exercise 15.10:** Write a sliding tile game. There are tiles numbered 1 to 15 randomly placed on a four by four grid with one empty space. If a tile adjacent to the empty position is clicked on, then it should move to that spot. When any other tile is clicked on nothing should happen.

**Exercise 15.11:** Write a checkers game that draws the board and the initial placement of the checker pieces. The user should be able to drag a piece from one square to another if it is a legal move. When the piece is released it should always be placed in the center of a square.

**Exercise 15.12:** Write an alarm program. When started the program should ask the user what time they want the alarm to go off and the message to display. At the specified time a window should open that displays the message in flashing colors.

**Exercise 15.13:** Exercise 6.22 involved the use of graphics to animate our earlier Television class, but even that exercise assumed that the standard method calls were used to manipulate a television instance. For this exercise, create a graphical user interface which resembles a remote control, with buttons for triggering behaviors such as channelUp and channelDown.