Chapter 6: UI Development Toolkit

Now, we have the basic understanding of how the interactive software operates with **events** (most of them predefined and sometimes newly defined by the user), **UI objects** and **event handlers**. The next question will be: "what are the **programming constructs** that allow us to specify these for actual implementation of a working interactive program? As was mentioned in the early part of Chapter 4 and from Figure 5.1, interactive programs and their interfaces are often developed using the **UI development toolkits (UI Toolkit for short)**. To be more precise, interfaces are developed using the UI toolkits and the the core computational logics using conventional programming languages. The UI toolkits are closely related to the UI Execution Framework (Chapter 5) on which the resulting interactive program would be running.

In a larger scheme of things, we can also think of a **UI development framework** (Chapter 7) as a methodology for the interactive program development as a whole. One such example might be a modular approach where the core computational and interface parts are developed separately and combined in a flexible manner. For one, this allows the concept of plugging in different interfaces for the same model computation and easier maintenance of the overall program. We first take a look at the UI toolkit.

6.1 User Interface Toolkit

The UI toolkit is a library of pre-composed UI objects (which would include event handlers as part of them) and a predefined set of events which are defined and composed from the lower level UI

software layer or the UI execution framework. The UI toolkit abstracts out the system details of handling events, and as such programming for interactive software becomes easier and more convenient. The UI object, since they often take the form of a manipulable graphical object the screen, is also often called as the **widget (i.e. window gadget)**. A typical widget set includes the menus, buttons, text boxes, images, etc. We have already examined typical widgets and UI objects in Chapter 4, which may be singular or composite (made up of several UI objects in themselves). The use of a toolkit also promotes the creation of an interface with a consistent look, feel and mechanism. Here, we take a closer look at the toolkits through three examples. In particular, we examine how events are defined, UI objects created, event handlers specified, and the interface (developed this way) combined with the core functional part of the program.

6.2 Java AWT UI Toolkit [4]

Java, as an object oriented language, offers a library of object classes called the AWT (Abstract Window Toolkit), which are classes useful for creating 2D UI and graphic objects. The "Component" is the most bare and abstract UI class from which other variant UI objects derive from. For instance, a window, button, canvas, label and others are descendants (subclasses) from the Component class. The "Window" class has further subclasses such as the "Frame" and "Dialog." Each class has basic methods, e.g. a window has such like resizing, adding sub-elements to it, setting its layout, moving to a new location, showing or hiding, etc. Figure 6.1 shows the overall UI object hierarchy and an example of the codes for creating a frame (a window

with a menu bar) and setting some of its properties by calling of such methods.

[Figure 6.1] The class hierarchy of UI objects (left) and creating a window and setting some of its properties through calling its class specific methods.

The Java AWT is not just a library of object classes for programming but also a part of the UI execution framework for Java which handles a (large) subset of interaction events (called the "AWTEvents"). In general, the interaction events are sent to Java programs which are captured, abstracted and stored as "EventObject." The AWTEvents are descendants of the EventObjects which cover most of the useful user interaction events (such as mouse clicks, keyboard input, etc.). The AWT framework will map the AWTEvent to the corresponding AWT UI object. There are two ways for the UI object to handle the events.

The first is by overriding the (predefined) callback methods of the interactive applet object for the events. Table 6.1 shows the AWTEvent "types" and the corresponding callback methods that can be overridden for customized event handling. Figure 6.2 shows a code example for reacting to various mouse events and redrawing the given interactive applet. Note that each event handler must return a Boolean value (true or false), indicating whether the event should be made available to other event handlers. If, however, the event cannot be processed for some reason, the event

handler may return false to signal that other components should process it.

[Table 6.1] AWTEvent types and corresponding overridable callback functions

AWTEvent type	Callback Function
mouseDown	mouseDown (Event evt, int x, int y)
mouseUp	mouseUp (Event evt, int x, int y)
mouseEnter	mouseEnter (Event evt, int x, int y)
mouseExit	mouseExit (Event evt, int x, int y)
mouseDrag	mouseDrag (Event evt, int x, int y)
gotFocus	gotFocus (Event evt, Object x)
lostFocus	lostFocus (Event evt, Object x)
keyDown	keyDown (Event evt, int key)
keyUp	keyUp (Event evt, int key)
action	Action (Event evt, Object x)

[Figure 6.2] An interactive applet with callbacks for mouse events. When the mouse click is entered, the applet draws an object at the click position. When the mouse moves, the whole applet is repainted and a new cursor is drawn at the newly moved position.

As a second mechanism for implementing reactive behaviors to various events, the individual AWT UI object is to be registered with an event listener that waits for and respond the corresponding event. The event handlers in Java AWT are known as the "listeners."[1] As its name suggests, it is a background process that listens for the associated events for a given UI object and responds to them. It is an abstraction of the event processing loop we have discussed in this section, assigned to a UI component for each various input events. Thus the listeners must be "registered" for various events that can be taken up by the given UI object. As a single UI object may be composed of several basic components and potentially receive many different types of input events, listeners for each of them will have to be coded and registered. Such a UI object is modeled as a collection of listeners through the Java "implementation interface" construct (see Figure 6.3).

[Figure 6.3] The event listener interface hierarchy in Java AWT.

Let's go back and take a look at the event component hierarchy (Figure 6.4). Table 6.2 shows the more detailed descriptions of the some of the various events. All the events derive from an abstract "EventObject" and offer basic methods for retrieving the object associated with the event

[1] To be precise, a "listener" is differentiated from a simple callback function because it is a process that waits for and react to the associated event, while a callback function is just the procedure that reacts to the event.

and accessing event type and id. Descendant event classes possess additional specific attributes and associated methods for accessing the values. For instance, the "KeyEvent" has a method called "getkeyChar" that returns the value of the keyboard input; the "MouseEvent" has a method called the "getPoint()" and "geClickCount()" that return the screen position data at which the mouse event occurred and the number of clicks. For more detailed information, the readers are referred to the reference manual for Java AWT [4].

[Figure 6.4] The event component hierarchy in Java AWT.

[Table 6.2] Java AWT event description and examples.

Event Class	**Description / Examples**
ActionEvent	Button press, double click on an item, selection of a menu item
AdjustmentEvent	Scroll bar movement
ComponentEvent	Hiding/revealing a component, component movement and resizing
FocusEvent	Component gaining or losing a focus
KeyEvent	Keyboard input
ItemEvent	Check box selection/deselection, menu item selection,
MouseEvent	Mouse button, mouse moving, mouse dragging, mouse focus

TextEvent	Text entry
WindowEvent	Window opening and closing, window de/activation, de/iconification

Just like the event component hierarchy, the event listener interface also is structured correspondingly as already seen in Figure 6.3. A UI object, possibly composed of several basic UI components, is designated to react to different events by associating the corresponding listeners with the UI object by the "interface implementation" construct. That is, the UI object is declared to "implement" the various necessary listeners and in the object initialization phase, the specific components are created and listeners registered. The class definition will thus include the implementation of the methods for the registered listeners. The named methods for the various listeners are illustrated in Table 6.3.

[Table 6.3] Events, corresponding listener interfaces, and derived methods in Java AWT.

Event Class	**Corresponding Listener** **Description**	**Samples of Derived Methods**
ActionEvent	ActionListener	actionPerformed
AdjustmentEvent	AdjustmentListener .	adjustmentValueChanged

ComponentEvent	ComponentListener	componentHidden componentMoved componentResized
FocusEvent	FocusListener	focusGained focusLost
KeyEvent	KeyListener	keyPressed keyReleased keyTyped
ItemEvent	ItemListener	itemStateChanged
MouseEvent	MouseListener	mouseClicked mouseEntered mouseExited mousePressed mouseReleased
MouseMotionEvent	MouseMotionListener	mouseDragged mouseMoved
WindowEvent	WindowListener	windowOpened windowClosed windowActivated

		windowDeactivated windowDeiconified windowIconified

An example of a UI object as an event listener implementation with some of its associated methods is shown in Figure 6.5. That is, "my_UI_ojbect" is declared as an extension of an "Applet" and at the same time as an implementation of two listeners (reacting to three types of events): the ActionEvent, and MouseEvent. It also has two button components created in the init(). Two listeners are also registered to each of the buttons (b1 and b2) in the same init() method. The method implementations follow.

[Figure 6.5] A UI object specification with event listeners.

6.3 Android UI Execution Framework and Toolkit [2]

The user programming environment and execution model for Android (even though at the low level, the operating system is derived from the LINUX) is based on Java. As such, the Android event processing model and programming toolkit structure are mostly the same as those of Java (or more specifically the Java AWT), except that the Android UI toolkit, in addition to the

programmatic method includes a "declarative" one for specifying the UI and defining their behaviors.

Events in Android can take a variety of different forms, but are usually generated in response to bare and raw external actions, such as touch and button input. Multiple or composite "higher" level events may also be internally recognized and generated like touch gestures (e.g. flick, swipe) or virtual keyboard inputs. The Android framework maintains an event queue into which events are placed as they occur. Events are then removed from the queue on a first-in, first-out (FIFO) basis. In the case of an input event such as touch on the screen, the event is passed to the "view" (the UI object classes in Android derive from what is called the "View" object) either by the location on the screen where the touch took place or by the current focus. In addition to the event notification, the view is also passed a range of information (depending on the event type) about the nature of the event such as the coordinates of the point of contact between the user's fingertip and the screen.

Similarly to the case of Java AWT, there are two major ways to define the reactive behavior to these events. The first is to override the default callback methods, similarly to those in Table 6.3 for Java AWT, of the "View" interactive class object for various typical input events.

[Figure 6.6] Overriding the default "View" methods such as the "onTouchEvent" and "onKeyDown" for defining interactive behaviors for the UI object "My_View" to the touch event and (virtual) keyboard inputs.

[Figure 6.7] "MyActivity" creates a View object called, "myView," and redefines "MyTouchListenerClass" by extending the the "View.OnTouchListener." Then an instance of "MyTouchListenerClass," "my_touchListener," is created and registered to the listener for "myView."

The second method is to associate an event listener with the "View" object." The Android View class, from which all user interface components are derived, contains a range of event listener interfaces, each of which contains an abstract declaration for a callback method. In order to respond to an event of a particular type, a view must register the appropriate event listener and implement the corresponding callback. For example, if a button is to respond to a mouse click event (the equivalent to the user touching and releasing the button view as though clicking on a physical button) it must both register the View.onClickListener event listener (via a call to the target view's setOnClickListener() method) and implement the corresponding onClick() callback

method. In the event that a "click" event is detected on the screen at the location of the button view, the Android framework will call the onClick() method of that view when that event is removed from the event queue. It is, of course, within the implementation of the onClick() callback method that any tasks should be performed or other methods called in response to the button click.

Figures 6.7~6.9 show three different (but in effect equivalent) ways to register and implement an event listener: (1) implementing the event listener itself and associating it with the view object (which is often part of the Android "Activity" object, Figure 6.7), (2) having the "view" object implement the event listener (Figure 6.8), and (3) having the top-most "Activity" object (which houses various view objects as its parts) implement the event listener (Figure 6.9).

[Figure 6.8] In this example, "My_Activity" creates "MyView" class by implementing the "View.OnTouchListner" and instantiates a "MyView" object "vw." The listener is registered by calling the "setOnTouchListener" on itself.

[Figure 6.9] In this example, "My_Activity" implements the View.OnTouchListener and thus includes and overrides the "onTouch" handler.

As already mentioned the Android UI framework also provides a "declarative" method for specifying the UI. That is, the form of the UI can be "declared" using a mark-up language (Figure 6.10). Through a development tool such as Eclipse [1], the UI can be built through direct graphical manipulation as well. Thus, in summary, there are three methods of UI development: (1) the usual programmatic way, (2) declarative way and (3) graphical way. In Eclipse, the de facto development tool for Android applications, any methods can be used and three methods can be used even simultaneously. Figure 6.10 shows the declarative UI specification ("main.xml" file) for "No Sheets" application designed in Chapter 4 through one of the sub-window in Eclipse.

[Figure 6.10] An example of a declarative specification of the UI (for the "No Sheets" application (Chapter 4). The declarative specification is saved in an XML file with element constructs for UI components such as image, buttons and etc. Note that the file names the elements (for later referral in the program) and one of the attribute of the UI object is the UI handler name (e.g. the encircled "load_file").

The corresponding UI can be displayed in the graphic form and be manipulated as shown in Figure 6.11. When a UI component is added, deleted, attribute values changed, such actions are reflected back to the respective representations, graphic or declarative. The Figures show that the UI screen is composed of an image and several buttons. The declarative specification names the event handler and other attribute values for the components in more exact terms. The handler code is implemented in the corresponding programmatic representation part.

[Figure 6.11]: An example of a graphic specification of the UI (for the "No Sheets" application (Chapter 4). The graphically displayed UI is consistent with the corresponding declarative representation (saved in the "main.xml").

[Figure 6.12] An example of a programmatic specification of the UI (for the "No Sheets" application (Chapter 4). Above shows the code implementation for the UI handers (e.g. "load_file") defined in the declarative specification.

6.4 Example: iOS UIKit Framework and Toolkit [3]

There are three major types of "discrete" events for iOS: Multi-touch, Motion and Remote Control (discrete events from external device such as remote controlled headphones). The iOS generates

low level events when users touch "views" of an application. The application sends these discrete events as UIEvent objects, as defined by the UIKit framework, to the view (i.e. specific UI component) on which the touches occurred. The view analyzes the touch event and responds to them. Touch events can be combined to represent higher level gestures such as flicks and swipes. The given application uses the UIKit classes for gesture recognition and responding to such "recognized" events. For continuous stream of sensor data such as those from accelerometers or gyroscopes, a separate "Core Motion" framework is used. Nevertheless, the mechanism is still similar in the sense that that sensor data, abstract event or recognized event is conveyed from iOS to the application then to the particular view according to the hierarchical structure of the application UI.

[Figure 6.13] iOS UIKit A UI object hierarchy example.

When users touch the screen of a device, iOS recognizes the set of touches and packages them in a UIEvent object that it places in the active application's event queue. If the system interprets the shaking of the device as a motion event, an event object representing that event is also placed in the application's event queue. The singleton UIApplication object managing the application takes an event from the top of the queue and dispatches it for handling. Typically, it sends the event to the application's key window (the window currently in focus for user events) and the window object representing that window sends the event to an initial object for handling.

[Figure 6.14] iOS UI Event Responder (handler) object class hierarchy [3].

That object is different for touch events and motion events. For touch events, the window object uses hit-testing and the "responder" chain to find the view to receive the touch event. In hit-testing, a window calls hitTest:withEvent: on the top-most view of the viewhierarchy; this method proceeds by recursively calling pointInside:withEvent: on each view in the view hierarchy that returns YES, proceeding down the hierarchy until it finds the subview within whose bounds the touch took place. That view becomes the hit-test view. If the hit-test view cannot handle the event, the event travels up the responder chain until the system finds a view that can handle it. For Motion and Remote Control events, the window object sends each shaking-motion or remote-control event to the first responder for handling. Although the hit-test view and the first responder are often the same view object, they do not have to be the same. The UIApplication object and each UIWindow object dispatches events in the sendEvent: method.

A responder object is an object that can respond to events and handle them. UIResponder is the base class for all responder objects, also known as, simply, responders. It defines the programmatic interface not only for event handling but for common responder behavior. UIApplication, UIView, and all UIKit classes that descend from UIView (including UIWindow) inherit directly or indirectly from UIResponder, and thus their instances are responder objects. The first

responder is the responder object in an application (usually a UIView object) that is designated to be the first recipient of events other than touch events. A UIWindow object sends the first responder these events in messages, giving it the first shot at handling them.

If the first responder or the hit-test view doesn't handle an event, UIKit may pass the event to the next responder in the responder chain to see if it can handle it. The responder chain is a linked series of responder objects along which an event is passed. It allows responder objects to transfer responsibility for handling an event to other, higher-level objects. An event proceeds up the responder chain as the application looks for an object capable of handling the event. Because the hit-test view is also a responder object, an application may also take advantage of the responder chain when handing touch events. The responder chain consists of a series of next responders (each returned by the nextResponder method) in the sequence. The response behavior itself is implemented by the responder objects (i.e. the UI components such as the window, button, slider, etc.).

[Figure 6.15] The event processing flow and the event driven object behavior structure. The user input is captured, abstracted and recognized by the UIKit framework and queued into the proper application view objects (or responders) who implement the particular response behaviors using the UIResponder methods.

6.5 Summary

In this chapter, we reviewed three examples of UI toolkits, namely, those for Java3D, Android, and iOS. There are certainly many other UI toolkits, however, most of them are similar in their structures and basic underlying mechanisms. As you have seen, some UI toolkits include visual prototyping tools and declarative specification syntax as well, which make it even more convenient for developers to implement user interfaces. In general, the use of UI toolkits promote standardization, familiarity, ease of use, fast implementation, and consistency for a given platform.

References

[1] Eclipse "Eclipse", http://www.eclipse.org/, (2013).

[2] Google Developer, "User Interface", http://developer.android.com/guide/ topics/ui/index.html, (2013).

[3] IOS Developer Library "iOS UIKit Framework Reference" https://developer.apple.com/library/ios/documentation/UIKit/Reference/UIKit_Framework/_index.html, (2013).

[4] Oracle, "Abstract Window Toolkit", http://docs.oracle.com/javase/7/docs/api/java/awt/package-summary.html, (2013).