Research on Event Handling Models of Java

WU Yue, WU Jing, ZHOU Ming-tian

(School of Computer Science and Engineering, UESTC, Chengdu 610054, China)

Abstract A new event-handling paradigm and its application model are proposed. The working mechanism and principle of event listener model is given in detail. Finally, the launching event mechanisms, the choosing event handling models and the dispatching mechanism are illustrated.

Keywords event handling models; event listener model; delegation model; dispatching

A software event indicates some action has occurred, to which a program might respond. Typical events includes keystrokes, mouse movement, button clicks, and messages receipts from the network, etc. Event handling is the ability to send and receive event notifications. Generally, there are four elements concerned in event handling:

- Event type: abstraction describing a set of related events;
- Event generator: object that generated the event;
- Event dispatcher: object that is responsible for dispatching the event to event handlers;
- Event handler: object that deals with the event.

An event may be generated by program or user action. The event generator, event dispatcher, or/and event handler may be the same object or different objects. The event handler may be one or more objects. The event handling can be specified as “when an event generator detects an event, the relevant event dispatcher notifies the related event handlers, then the event handlers respond the event.”

There are two famous event handling models[1]. One is called inheritance model, which is based on inheritance. Another one is called event listener model, or delegation event model. Inheritance model is used in JDK 1.0. Events are encapsulated in a single class named Event. Only AWT component objects could handle events in the handleEvent() method which is called action() method. For catching and handling events, the component that handles an event has to be either the component in which the event occurred or a component above it in the component containment hierarchy. The handleEvent() method either handles events and returns true, or false or super, handleEvent() to propagate them up the containment hierarchy. So there are two choices for handling an event in the following:

- Let each subclass of AWT components handling its events by overriding either action() or handleEvent() methods and returning true;
- Let a particular container handle all events for its subclasses.

Event Listener model used in JDK 1.1 is considered as one of the greatest significant changes from JDK 1.0 to JDK 1.1, and it has many advantages over inheritance model[2], such as more flexible, more feasible, more robust, etc. and can filter events. But it is also often intricate and difficult to understand. So in this paper, we introduce a new event-handling paradigm and propose a application model of event handling for Java. In section 1, We detail the working mechanism and principle of event listener model; In section 2, the launching event mechanisms, the choosing event handling models and the dispatching mechanism are illustrated.

1 Event Listener Model

1.1 JDK 1.1 Event Listener Model Working Mechanism

We can outline the JDK event listener model as
shown in Fig. 1. We use Eventtype to present an event type. The JDK 1.1 event handling model works basically as follows:

When an event happens, it is first posted to the system event queue which is monitored by an AWT thread (EventDispatchThread);

Then the dispatchEvent() method is called by the AWT thread;

Then the processEvent() method is called in the which 1.1 event handling model or 1.0 event handling model is chosen. If any kind of listener is registered or any kind of event type is explicitly enabled, then JDK 1.1 event handling model is taken by calling appropriate processEventtypeEvent() method, otherwise the event is treated as Inheritance Model by calling handleEvent() method.

In JDK 1.1 event handling model, the processEvent() method by default will invoke the appropriate processEventtypeEvent() method which deals with the event of specific event type indicated by Eventtype, such as invoking processMouseEvent() method for mouse event.

By default, each processEventtypeEvent() method will invoke any registered listener. At here, the Event Listener Model starts to work, e.i., let event dispatcher to deliver the event to all registered event handlers.

Fig. 1 JDK1.1 Event Handling Model

From Fig. 1, we can see there are four choices for handling an AWTEvent:

1. Overriding the processEvent() method. As all events of AWTEvents are first sent to the super.handleEvent() method which is declared in java.awt.component. So we can override it to deal with some events, and call the super.processEvent() method for handling other events.

2. Overriding the processEventtypeEvent() method. Generally, when there is any kind of listener registered or any kind of event type explicitly enabled, processEvent() will invoke proper processEventtypeEvent(). So we can also at here override the processEventtypeEvent() method to deal with specific event types. Meanwhile, remember to call the super.processEventtypeEvent() for handling other event.

3. Using JDK 1.0 event handling model by overriding the handleEvent() or action() method.

4. Using JDK 1.1 event listener model. If there is any kind of listener registered and if we have not taken the first two choices, the event listener model starts to dispatch the event to all registered listeners, and let the listeners to handle the event.

1.2 Event Listener Model Principle

In event listener model, the event generator is called event source, the event handler is called event listener. The role of the event dispatcher is taken by a class called AWTEventMulticaster, which manages an immutable structure consisting of a chain of event listeners and dispatches events. We use EventSource to present Event Source class, a EventSource present an object of EventSource. EventtypeListener presents an object of EventtypeListener.

The event listener model is also called delegation event model. The event source object delegates the event listener object to handle the event, e.i., the event will be generated by the event source object but caught by the event listener object. Objects of any type can be registered as event listeners. Event listeners receive notification only about the Eventtypes they have been registered. The listener mechanism is implemented by calling a addEventtypeListener(aEventListener) method of EventSource which has two functions: first, notifying that it will take 1.1 event listener model; second, creating a list which registers the event listener.
object with event source object.

The relation between EventSource and EventType is many-to-many, i.e., one EventSource object can generate many kinds of EventType and one EventType can be fired by many EventSource objects. The relation between EventType and EventListener is one-to-many, i.e., one EventType can have many EventListener interfaces or classes, but each EventListener can only handle one kind of EventType. The relation between EventSource and EventListener is many-to-many. It means that one EventSource object can delegate many EventListener objects to handle its event; vice versa, one EventListener object can also be registered with many EventListener objects.

The event listener mechanism can be described as follows:

First, EventListener objects are registered with EventSource object by using addEventTypeListener() method. Then, when the event source object generates one event with the type of EventType, this event is dispatched to all EventListener objects registered for this EventSource object. Finally, this event is handled by calling the methods defined in registered EventListeners.

Event listener model is classified as reserved event listener model (RELM) and extended event listener model (XELM). The difference between them is that there exists event listener interfaces which can be directly used in RELM, but in XELM there is not existing event listener interfaces which need to be extended.

2 Application of Event Handling Model

In this section, there are three points to consider when using JDK 1.1 event handling model, i.e. how to launch an event, how to choose the event handling model, and how to implement the EventDispatcher, etc.

2.1 Launching Event Mechanisms

An event can be generated by user, such as tapping a key from keyboard, clicking mouse, etc. or by program, such as when some data is received from network, then the program fires disconnecting network event. We can also fire an event by another event, such as, when mouse is pressed in a specific area, then a new event can be generated by this mouse event.

For Reserved Eventtypes, there exist corresponding methods to capturing and handle their events. The details of who, how and when calling these methods are transparent to programmer. But for new Eventtypes, the way for capture new EventType events generally is with the help of an AWT component when its specific event (called helper event) occurs, it fires the new EventType event. For example, when mouse is pressed in a specific area, or when a specific button is pressed by clicking the mouse, the MouseEvent.MOUSE_PRESSED event is generated in both cases. As we have known, all AWT event is first passed to processEvent() method and then to processMouseEvent() method, we can override either processEvent() method or processMouseEvent() method to identify the helper event, then to launch the new EventType event.

Launching an event has three ways indicated with ①, ②, ③ in Fig.1.

① Posting event into SystemEventQueue: The AWTEvent posted in SystemEventQueue will be extracted by the AWT event thread and be delivered later by calling dispatchEvent() method.

② Calling dispatchEvent() method immediately to invoke processEvent() method.

③ In processEventTypeEvent() method, dispatching directly event to all registered listeners that begin to handle the event.

At first, we identify helper event always in processhelperEvent() method. For the below example, we can override the processMouseEvent() method for identifying the specific MOUSE_PRESSED event happened, then launching a new EventType event. The code may be as follows:

```java
protected void processMouseEvent (MouseEvent e) {
    if (e.getID()==MouseEvent.MOUSE_PRESSED){
        // Identifying the specific event, such as using x,y to decide whether in the specific area,
        // or using button name to distinguish different buttons.
    }
}
```
// Launching a new Eventtype event choosing one of the below three ways:
// dispatching directly to registered listeners
// list of eventtypeListener is created by using addEventtypeListener() method
if (eventtypeListener != null) {
    Eventtype eventtype = new Eventtype(this);
eventtypeListener.handlingEvent(eventtype);
    // handlingEvent() is one method of listener, such as event0(), event1().
} // end if (eventtypeListener != null)
// Calling dispatchEvent() method immediately to invoke processEvent() method
dispatchEvent (new Eventtype (this));
// Posting event into SystemEventQueue
Toolkit aToolkit = getToolkit();
EventQueue anEventQueue = aToolkit.getSystemEventQueue();
AEventQueue.postEvent (new Eventtype (this));
} // end of (e.getID() == MouseEvent.MOUSE_PRESSED)
// An important point that should be remembered is to call super.processMouseEvnet()
super.processMouseEvent (e);
} // end processMouseEvent

2.2 Choosing Event Handling Models

2.2.1 The Simplest Way is Using the Reserved Event Listener Model

When using reserved event listener Eventtype, many things are automatically done by Java API classes, such as from calling processEvent(), processEventtypeEvent() methods to dispatching event to all registered listeners, and all the relations and supporting classes for using event listener model are already created, such as Eventtype, EventtypeListener interfaces, addEventtypeListener() methods, and EventDispatcher.

All these details are transparent to programmer. The minimum things needed to be done are: First, defining event listener class and overriding the appropriate methods to handling event; Then, using addEventtypeListener() to register event listener objects with event source object; Finally, invoking event listener method for handling event. So a good rule of thumb is that if one of reserved event listener Eventtypes can be used, the best and the simplest way is to directly use it. Don’t try to extend a new Eventtype, or override processEvent() or any processEventtypeEvent() method.

2.2.2 The Extend Way

If we have to extend a new Eventtype for using event listener model, the following aspects are needed to be considered:

Subclass one of AWT components as the helper event. By doing so, we can easily identify this helper event in its processhelperEvent() method, then launch the new Eventtype event by using appropriate way.

According to section 1, defining corresponding supporting classes, such as Eventtype class, EventtypeListener interface, EventtypeAdapter class, EventSource class include addEventtypeListener() and removeEventtypeListener() methods, EventListener class include the method for handling event, and the EventDispatcher class.

Register listener objects with source object, launch a new Eventtype event, dispatch it to all registered listeners and let them to handle it.

2.2.3 Without Using Listener Model

We can always handle an event at where it is identified. For example, if the event is a Reserved event type, it must be passed to processEvent() method, we can override the processEvent() method and use some methods already defined in Java API classes to identify the event, then program to handle it. We can also let it be passed to processEventtypeEvent() method, then identify it and handle it. If the event is not a Reserved event type, we can use a helper event to launch this new Eventtype event[5]. So we first override the processhelperEvent() method for identifying the helper event occurred, then program to handle the new Eventtype event in processhelperEvent() method. Then, if we take the way of 1 or 2, the new Eventtype event has not been handled yet. It can be passed to processEvent(). Note that processEvent(AWTEvent e) method accepts only AWTEvent object, so the new Eventtype class processEvent() method for dispatching the new Eventtype event to registered listeners. The
protected oid processEvent(AWTEvent e) {
    // identify the new Eventtype event
    if (e instanceof Eventtype) {
        processEventtypeEvent((Eventtype) e);
    } else {
        super.processEvent(e);
    } // end processEvent()

protected synchronized void processEventtypeEvent(
    Eventtype e) {
    if (eventtypeListener != null )
        eventtypeListener.handlingEvent (e);
    // handling is one method of listener,
    //such as event0(), event1().
} // end processEventtypeEvent()

2.3 Understanding the Dispatching Mechanism

As one EventSource object can delegate many
EventListener objects for handling its event[6], so the
main task of EventDispatcher is how to manage all
registered event listeners and dispatch the event to
them. In generally, there are three approaches to
implement EventDispatcher.

2.3.1 Using AWTEventMulticaster Class as
EventDispatcher for reserved AWTEvent Types
when using the Reserved AWTEvent types,
AWTEventMulticaster class plays the EventDispatcher
role, and the EventDispatcher is completely transparent
to programmer. AWTEventMulticaster can also be
used as EventDispatcher for adding new
addEventtypeListener() method by calling
AWTEventMulticaster.add() method. In fact,
AWTEventMulticaster class is also a listener for all
reserved AWTEvent types, as it implements all 11
reserved EventListener interfaces.
AWTEventMulticaster class uses a “Binary_tree” data
structure to manage the registered event listener
objects for each event type. Each leaf node refers to an
event listener object. In this Binary_tree, all registered
event listener objects are corresponding to all leaves.
And the root is referred by a EventtypeListener
variable (such as actionListener). Dispatching the
event to all registered event listener objects is just like
traversing the Binary_tree.

2.3.2 Creating new EventtypeEventMulticaster Class
as EventDispatcher for New Eventtypes
EventtypeEventMulticaster class must implement
EventListener interface. So all methods in
EventListener interface must be implement
EventtypeEventMulticaster class. For supporting add
EventListener() and removeEventListener() methods, add() and remove() methods must also
implemented in EventtypeEventMulticaster class. For example :
Class NewEventtypeMulticaster implements
NewEventListener {
    Protected NewEventListener a,b;
    Protected NewEventtypeEventMulticaster
    (NewEventListener a, NewEventListener b){
        this.a = a;
        this.b = b;}
    static NewEventtype add (NewEventtype a,
    NewEventListener b) {
        if ( a = = null ) return b;
        else if ( b = = null ) return a;
        else return newEventtypeMulticaster(a,b);
    } // end add()
    static NewEventListener remove
    (NewEventListener a, NewEventListener b ) {
        if ( a = = null ) || (a = = b) return null;
        else if ( a instanceof NewEventtypeMulticaster)
            return add
        remove((( NewEventtypeEven
tMulticaster) a).a, b);
        else return a;
    } // end remove()
    // overriding all methods in
    NewEventListener interface
    public void event0 ( NewEventtype e) {
        a.event0 (e); b.event0 (e);}
    public void event1 ( NewEventtype e) {
        a.event1 (e); b.event1 (e);}
    } // end NewEventtypeMulticaster

2.3.3 Using Vector to Manage Multiple Listeners
In EventSource class, a Vector for managing
multiple listeners is defined, addEventListener() and
removeEventListener() methods are also
defined by using addElement() and removeElement() methods of Vector, and the processEventtypeEvent()
A method is implemented for dispatching the event to all registered listeners in Vector. For example:

```java
protected Vector aVector = new Vector();
// using aVector to manage multiple listeners
public void addNewEventTypeListener ( NewEventTypeListener, listener) {
    aVector.addElement ( listener);
} // end addNewEventTypeListener()
public void removeNewEventTypeListener ( NewEventTypeListener, listener) {
    aVector.removeElement ( listener);
} // end removeNewEventTypeListener()
protected void processNewEventTypeEvent (NewEventType e) {
    synchronized (aVector) {
        for ( int elem = 0; elem < aVector.size(); elem ++)
            (( NewEventTypeListener) aVector.elementAt (elem)).handlingEvent (e);
        // handlingEvent() is one method of listener, such as event0(), event1() })
} // end processNewEventTypeEvent()
```

### 3 Conclusions

Efficient event handling is the key to creating powerful, GUI-based Java software. Event handling in JDK 1.1 is based on the concepts of event listeners and notifiers. For them to communicate properly, an event object and listener interface are used. This model provides a robust and extensible capability which can be applied to non-visual events as well as GUI ones.

### References


### Brief Introduction to Author(s)

**WU Yue** (吴跃) was born in 1958. He received B.S. and M.S. from Harbin Institute of Technology in 1982 and 1984, respectively, majoring in computer science and engineering. He has been working in the Department of Computer Science and Engineering of UESTC since 1984. He was awarded the National Research Fellowship and visiting scholar to University of Geneva from 1991 to 1993 and to University of Paris 1 from 1997 to 1998 respectively. He is currently the Dean of the Computer Science and Engineering College and the Dean of the Computer Software College of UESTC. He has co-authored several papers in some journal and conferences. The current research fields cover distributed object computing, Java and mobile agent technologies.

**WU Jing** (吴劲) was born in 1972. She received B.S. and M.S. from UESTC in 1993 and 1996 respectively. She is now pursuing the Ph.D. degree and at the same time she works as a lecturer at UESTC. Her current research interests include: computer network and distributed database.