Event-Based Programming – Part II

Rob Miller & Saman Amarasinghe
Fall 2007
Today’s lecture

Model-view-controller pattern
- Found throughout user interfaces

Sequential vs. Event-based programming

Pitfalls of Event-based programming

Concurrency in Even-based Programming

Handling Events that Block

© Robert Miller 2007
View Hierarchy

**Hotspot data structure is better represented as a tree**

- Each object in the tree is a **view** (aka component, widget, control, interactor, element)
- Each view has a **bounding box** representing the screen area it occupies
- A child view’s bounding box is nested inside its parent’s bounding box

© Robert Miller 2007
**Input Handling**

**Input handlers are associated with views**

- Handlers are also called **listeners**, event handlers, subscribers, and observers (not to be confused with observer methods in an ADT!)
Event-Based Programming

Control flow through a graphical user interface

- Top-level loop (event loop) reads all input from mouse and keyboard
  - Event loop finds the appropriate view in the view hierarchy (by looking at x,y position) and calls its listener(s)
- Listener changes state of the interface (e.g. modifying the view hierarchy) and returns immediately to the event loop
  - e.g. doAdd() creates a new dialog box, attaches them to the view hierarchy, but doesn’t wait for the actual editing input – it attaches listeners on the textboxes to take care of that
Component is very weakly coupled to its listeners

- Component doesn’t depend on the identity of the listening class, only that it implements the MouseListener interface
- Component doesn’t depend on the number of listeners, and listeners can come and go
Publish-Subscribe Pattern

GUI input handling is an example of the Publish-Subscribe pattern

- aka Listener, Event, Observer

An event source generates a stream of discrete events

- In this example, the mouse is the event source
- Events are state transitions in the source
- Events often include additional info about the transition (e.g. x,y position of mouse), bundled into an event object or passed as parameters

Listeners register interest in events from the source

- Can often register only for specific events – e.g., only want mouse events occurring inside rectangle
- Listeners can unsubscribe when they no longer want events

When an event occurs, event source distributes it to all interested listeners
Other Examples of Publish-Subscribe

**Higher-level GUI input events**
- A Button sends an action event when it is pressed (whether by the mouse or by the keyboard)
- A Textbox sends change events when its contents change

**Internet messaging**
- Email mailing lists
- IM chatrooms
- RSS feeds
We’ve seen how to separate input and output in GUIs

- Output is represented by the view hierarchy
- Input is handled by listeners attached to views

Missing piece is the backend of the system

- Backend (aka model) represents the actual data that the user interface is showing and editing
- Why do we want to separate this from the user interface?
public class PreviewPane {
    public void display(Set<Photo> s) {
        m_currentThumbnails = new ArrayList<Thumbnail>();
        u_content = new JPanel();
        for (Photo p: s) {
            Thumbnail o = new Thumbnail(p);
            o.addMouseListener(new SelectionToggleListener(o));
            u_content.add(o);
            m_currentThumbnails.add(o);
        }
        u_content.repaint();
    }
}

private class SelectionToggleListener extends MouseInputAdapter {
    private final Thumbnail m_thumb;
    public SelectionToggleListener(Thumbnail t) {
        m_thumb = t;
    }
    @Override
    public void mouseClicked(MouseEvent e) {
        m_thumb.toggle();
        m_thumb.repaint();
    }
}
Photo Organizer Application
Model-View-Controller Pattern

View handles output
- calls observers on the model to display it
- listens for model changes and updates display

Controller handles input
- listens for input events on the view hierarchy
- calls mutators on model or view

Model maintains application state
- implements state-changing behavior
- sends change events to views

© Robert Miller 2007
Model-View-Controller Pattern

View handles output
• calls observers on the model to display it
• listens for model changes and updates display

Controller handles input
• listens for input events on the view hierarchy
• calls mutators on model or view

Model maintains application state
• implements state-changing behavior
• sends change events to views

© Robert Miller 2007
Textbox Example

JTextField is a Component that can be added to a view hierarchy

KeyListener is a listener for keyboard events

JTextField

KeyListener

KeyPress events

Document

KeyListener

Move cursor

Text change events

Get text

Edit text

Document represents a mutable string of characters

© Robert Miller 2007
Textbox Example

JTextField is a Component that can be added to a view hierarchy.

KeyListener is a listener for keyboard events.

KeyListener

KeyPress events

move cursor

document

get text

edit text

Document represents a mutable string of characters.

© Robert Miller 2007
Advantages of Model-View-Controller

Separation of responsibilities
- Each module is responsible for just one feature
  - Model: data
  - View: output
  - Controller: input

Decoupling
- View and model are decoupled from each other, so they can be changed independently
- Model can be reused with other views
- Multiple views can simultaneously share the same model
- Views can be reused for other models, as long as the model implements an interface
  - e.g. JList class (the view) and ListModel interface

© Robert Miller 2007
Selecting Photos: Sequential vs. Event based

```java
Case (action) {
    .....  
    switch SELECT_PICTURES:
    private Thumbnail m_thumb = SelectPicture();
    while(m_thumb) {
        m_thumb.toggle();
        m_thumb = SelectPicture();
    }
}
```
Selecting Photos: Sequential vs. Event based

```java
Case (action) {
    ......
    switch SELECT_PICTURES:
        private Thumbnail m_thumb = SelectPicture();
        while(m_thumb) {
            m_thumb.toggle();
            m_thumb = SelectPicture();
        }
    }
}

private class SelectionToggleListener extends MouseInputAdapter {
    private final Thumbnail m_thumb;
    public SelectionToggleListener(Thumbnail t) {
        m_thumb = t;
    }
    @Override
    public void mouseClicked(MouseEvent e) {
        m_thumb.toggle();
        m_thumb.repaint();
    }
}
```

© Robert Miller 2007
Selecting Photos: Sequential vs. Event based

**Sequential**
- Pros: Simple control-flow
- Cons: Users have to exactly follow a sequence

![Sequential Flowchart]

**Event based**
- Pros: Users have the freedom
- Cons: Spaghetti control-flow

![Event based Flowchart]
Risks of Event-Based Programming

Spaghetti of event handlers

- Control flow through an event-based program is not simple
- You can’t follow the control just by studying the source code, because control flow depends on listener relationships established at runtime
- Careful discipline about who listens to what (like the model-view-controller pattern) is essential for limiting the complexity of control flow

Obscured control flow leads to some unexpected pitfalls...
Basic Interaction of Event Passing

Sequence diagram is good for depicting control flow

- Time flows downward
- Vertical time lines represent objects
- Horizontal arrows show method calls and returns passing control between objects
- Dark rectangles show when a method is active (i.e., on the stack)
Basic Interaction of Event Passing

**Sequence diagram is good for depicting control flow**

- Time flows downward
- Vertical time lines represent objects
- Horizontal arrows show method calls and returns passing control between objects
- Dark rectangles show when a method is active (i.e., on the stack)

```java
interface Source {
    addListener()
    removeListener()
    get()
    set()
}

interface Listener {
    changed()
}
```
Basic Interaction of Event Passing

**Sequence diagram is good for depicting control flow**

- Time flows downward
- Vertical time lines represent objects
- Horizontal arrows show method calls and returns passing control between objects
- Dark rectangles show when a method is active (i.e., on the stack)

```java
interface Source {
    addListener()
    removeListener()
    get()
    set()
}

interface Listener {
    changed()
}
```

© Robert Miller 2007
Basic Interaction of Event Passing

**Sequence diagram is good for depicting control flow**

- Time flows downward
- Vertical time lines represent objects
- Horizontal arrows show method calls and returns passing control between objects
- Dark rectangles show when a method is active (i.e., on the stack)

```java
interface Source {
    addListener()
    removeListener()
    get()
    set()
}

interface Listener {
    changed()
}
```

© Robert Miller 2007
Pitfall #1: Listener Calls Observers

The listener often calls methods on the source

- e.g., when a textbox gets a change event from its model, it needs to call `getText()` to get the new text and display it
- So calls to `get()` may occur while `set()` is still in progress
- Why is this a potential problem?
Pitfall #1: Listener Calls Observers

The listener often calls methods on the source

- e.g., when a textbox gets a change event from its model, it needs to call `getText()` to get the new text and display it
- So calls to `get()` may occur while `set()` is still in progress
- Why is this a potential problem?

Solution: source must establish its rep invariant before giving up control to any listeners

- Often done simply by waiting to send events until end of `set()`
Pitfall #2: Listener Calls Mutators

The listener might call set() on the source

➤ This rarely happens directly; more often indirectly, when two models are listening to each other in order to keep their state synchronized
➤ So calls to set() may occur while set() is still in progress
➤ Why is this a potential problem?
Pitfall #2: Listener Calls Mutators

The listener might call set() on the source

- This rarely happens directly; more often indirectly, when two models are listening to each other in order to keep their state synchronized.
- So calls to set() may occur while set() is still in progress.
- Why is this a potential problem?

Client | Model 1 | Listener 1->2 | Model 2 | Listener 2->1
Pitfall #2: Listener Calls Mutators

The listener might call set() on the source

- This rarely happens directly; more often indirectly, when two models are listening to each other in order to keep their state synchronized.
- So calls to set() may occur while set() is still in progress.
- Why is this a potential problem?

- Solution: only send events when set() actually causes a change in the source.

© Robert Miller 2007
Pitfall #3: Listener Removes Itself

The listener might call removeListener() on the source

- This happens when the listener is done its work, e.g. a listener that executes a stock trade as soon as a certain price is reached
- So calls to removeListener() may occur while set() is still in progress
- Why is this a potential problem?
Pitfall #3: Listener Removes Itself

The listener might call removeListener() on the source

- This happens when the listener is done its work, e.g. a listener that executes a stock trade as soon as a certain price is reached.
- So calls to removeListener() may occur while set() is still in progress.
- Why is this a potential problem?

Solution: send events by iterating over a copy of the listeners data structure, or use javax.swing.EventListenerList which copies when needed.
Events from Everywhere

Mouse and keyboard events happen asynchronously

Mouse thread

Mouse

Keyboard thread

Keyboard

main thread

Main

© Robert Miller 2007
Java Swing Is Not Threadsafe

The view hierarchy is a big meatball of shared state

- And there’s no lock protecting it at all
- It’s OK to access user interface objects from the event-handling thread (i.e., in response to input events)
- But don’t touch – read or write – any Component objects from a different thread
- Even using Swing from the main thread of the program is skating on thin ice:
  ```java
  public static void main(String[] args) {
    JFrame frame = new JFrame();
    frame.setVisible(true);
    frame.setTitle("My Window");
  }
  ```
  risky code, because it’s using Swing from the main thread, not the event-handling thread

- See “Threads and Swing”,
Concurrency in GUIs

How to Enforce safety of the GUI objects

- Thread-confined?
- Single owner?
- Shared read-only?
- Shared thread-safe?
- Guarded by others?
Concurrency in GUIs

Mouse and keyboard events are accumulated in an event queue

- Event loop reads an input event from the queue and dispatches it to listeners on the view hierarchy
- In Java, the event loop runs on a special event-handling thread, started automatically when a user interface object is created

© Robert Miller 2007
The event queue is also a message-passing queue

- To access or update Swing objects from a different thread, you can put a message (represented as a Runnable object) on the event queue.
  
  ```java
  SwingUtilities.invokeLater(new Runnable() {
      public void run() {
          frame.setTitle("...");
      }
  });
  ```

- The event loop handles one of these pseudo-events by calling `run()`.
Events that Block

```java
private final JTextField info;

public void updateInfo(String str) {
    info.setText(str);
    info.repaint();
}

calcBlock.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent e) {
        System.out.println("Starting calc");
        updateInfo("start calc");
        try {
            Thread.sleep(5000);
            updateInfo("done calc.....");
            System.out.println("Done calc");
        } catch(Exception ex) { }
    }
});
```

Event handler runs too long
➤ GUI is frozen

© Robert Miller 2007
Events that Block

calcBack.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent e) {
        System.out.println("Swing: Starting calc");
        updateInfo("start calc");
        (new Thread() {
            public void run() {
                try {
                    System.out.println("Thread: start calc");
                    Thread.sleep(5000);
                    updateInfo("done calc.....");
                    System.out.println("Thread: Done calc");
                } catch (Exception ex) { }
            }
        }).start();
        System.out.println("Swing: Done calc");
    }
});

Start the long task in a separate thread

➤ Return the GUI thread back so it can process the next event
Events that Block

calcBack.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent e) {
        System.out.println("Swing: Starting calc");
        updateInfo("start calc");
        (new Thread() {
            public void run() {
                try {
                    System.out.println("Thread: start calc");
                    Thread.sleep(5000);
                    SwingUtilities.invokeLater(
                        new Runnable() {
                            public void run() {
                                updateInfo("done calc.....");
                            }
                        });
                    System.out.println("Thread: Done calc");
                } catch(Exception ex) { }
            }
        }).start();
        System.out.println("Swing: Done calc");
    }
});

But…need to be careful about synchronization

© Robert Miller 2007
Summary

**Publish-subscribe pattern**
- An event source sends a stream of events to registered listeners
- Decouples the source from the identity of the listeners
- Beware of pitfalls

**MVC pattern**
- Separation of responsibilities: model=data, view=output, controller=input
- Decouples view from model

**Concurrenty in GUIs**
- Message queue handles multiple inputs
- Can be used to handle events that block
  - Be careful! In Swing, use SwingUtilities.invokeLater()
public class PhotoOrganizer extends JFrame {
    private List<Thumbnail> m_currentThumbnails;
    ...
    calcBack.addActionListener(new ActionListener() {
        public void actionPerformed(ActionEvent e) {
            WWW
            (new Thread() {
                XXX
                public void run() {
                    YYY
                    SwingUtilities.invokeLater(
                        new Runnable() {
                            public void run() {
                                ZZZ
                            }}}}));
            }}).start();
        }
    }
}

Synchronization Requirements
In the above code fragment from PhotoOrganizer, what are the requirements on methods and members of
the class that are used in locations marked WWW, XXX, YYY, and ZZZ.

Counting people on photos
int peopleCount = 0;
for (Thumbnail t: m_currentThumbnails) {
    peopleCount = peopleCount + t.people();
    updateInfo(peopleCount);
}

Assume an expensive method that identify and count the number of people in a thumbnail is provided. How
do you add the above code fragment to the CalcBack listener? (show the code fragments for WWW, XXX,
YYY and ZZZ.) Note that m_currentThumbnails is a mutable list, while Thumbnail is an immutable type.
public class PhotoOrganizer extends JFrame {

    private List<Thumbnail> m_currentThumbnails;

    calcBack.addActionListener(new ActionListener() {
        @Override
        public void actionPerformed(ActionEvent e) {

            new Thread() {
                @Override
                public void run() {

                    SwingUtilities.invokeLater(
                            new Runnable() {
                                public void run() {

                                    int peopleCount = 0;

                                    for (Thumbnail t: m_currentThumbnails) {
                                        peopleCount = peopleCount + t.people();
                                        updateInfo(peopleCount);
                                    }
                                }
                            });
                    }
                }
            }).start();
        }
    });

    public void actionPerformed(ActionEvent e) {

        (new Thread() {
            @Override
            public void run() {

                SwingUtilities.invokeLater(
                        new Runnable() {
                            public void run() {

                                int peopleCount = 0;

                                for (Thumbnail t: m_currentThumbnails) {
                                    peopleCount = peopleCount + t.people();
                                    updateInfo(peopleCount);
                                }
                            }
                        });
                    }
                }
            }).start();
        }
    }