Event-Based Programming

Rob Miller & Saman Amarasinghe
Fall 2007
Today’s lecture

Event-based programming
➤ Example: input handling in graphical user interfaces

Composite pattern
➤ Example: view hierarchy in GUIs

Model-view-controller pattern
➤ Found throughout user interfaces
Composing a Complex UI
View Hierarchy

Liberal use of containers

Simple layout within each container
- Size, boarders, layout rules (X_AXIS, Y_AXIS, LINE_AXIS etc.)

View hierarchy is the fundamental structuring pattern for graphical user interfaces
- for input (by attaching listeners)
- for output (paint() method draws the views recursively),
- automatic layout (containers recursively size and position their children)

© Robert Miller 2007
View Hierarchy Object Model

View hierarchy is an example of the Composite pattern

- Composite allows groups of objects (e.g., Panel, Window) to be used in the same way as primitive objects (e.g., Button, Text, Label) – i.e., primitives and containers are both subsets of Component
Handling Mouse Input

Consider an Photo Organizer application

How to handle mouse input?

- A simple command handler (like you may have written in the Project 2 chat server) might do it like this:

```java
while (true) {
    read mouse click
    if (clicked on Add) doAdd();
    else if (clicked on Delete) doDelete();
    else if (clicked on Thumbnail) doToggle();
    ...
}
```

© Robert Miller 2007
Mouse Targets Change Frequently

Suppose the user clicks Edit...

```
void doEdit() {
    change window to show Name and Email textboxes
    while (true) {
        read mouse click
        if (clicked on Name textbox) selectName();
        else if (clicked on OK) doOK();
        else if (clicked on Cancel) doCancel();
        else if (clicked on Add) doAdd();
        else if (clicked on Remove) doDelete();
    }
```

In general, wherever we are in the program, we need handle clicks on any visible target
Decouple Input Handling

Key idea: represent the input-handling code as a data structure

- Here’s the simplest data structure we might imagine: a list of hotspots on the screen where a mouse click causes the program to do something

\[
\text{hotspots} : (\text{Rectangle} \times \text{Handler})^* \\
\text{Rectangle} = \text{int}^4 \\
\text{Handler} = \text{void} \rightarrow \text{void}
\]

\[
\text{hotspots} = ((\text{Add}, \text{doAdd}), \\
(\text{Delete}, \text{doDelete}), \\
(\text{Glass Buildings}, \text{selectName}), \\
...)
\]

- Now the input handling code simply looks like this:

```plaintext
read mouse click
for each hotspot \( \in \) hotspots {
    if (clicked in hotspot.rectangle) hotspot.handler()
```

© 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
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
A Closer Look at Listeners

```
class Component {
    ...
    public void addMouseListener(MouseListener l) ...
    public void removeMouseListener(MouseListener l) ...

    private void fireMousePress(int x, int y) {
        MouseEvent event = new MouseEvent(..., x, y, ...);
        for (MouseListener l : listeners) {
            l.mousePressed(event);
        }
    }
}
```

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

© Robert Miller 2007
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 a rectangle
- Listeners can unsubscribe when they no longer want events

When an event occurs, event source distributes it to all interested listeners

© Robert Miller 2007
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
Listening to a Backend Model

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

View

PreviewPlane

Controller

SelectionToggleListener

Model

Photo

Thumbnail
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
Photo Organizer Application

View
- PreviewPlane

Model
- Photo
- Thumbnail

Controller
- Listener
- SelectionToggleListener
Textbox Example

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

KeyListener is a listener for keyboard events

KeyPress events

move cursor

text change events

get text

Document represents a mutable string of characters

© Robert Miller 2007
Photo Organizer Application

View
- PhotoOrganizer
- CatalogTreeModel
- JButton
- PreviewPlane
- Thumbnail

Controller
- ActionListener
- SelectionToggleListener
- TreeModelListener

Model
- Photo
- Collection
- Catalog

Diagram showing the relationships between the View, Controller, and Model components, with arrows indicating the flow of information and interactions between the different parts of the application.
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
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)

Client

Source

Listener

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

interface Listener {
    void 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?

```
Client          Source          Listener
```

© 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?

![Diagram showing the interaction between Client, Source, and Listener]

- 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?

Solution: only send events when set() actually causes a change in the source
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.EventLister.List which copies when needed
Concurrent 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
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”,

© Robert Miller 2007
Message Passing Via the Event Queue

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()
Summary

View hierarchy
- Organizes the screen into a tree of nested bounding boxes
- Used for dispatching input, as well as displaying output
- Uses the Composite pattern: compound views (windows, panels) can be treated just like primitive views (buttons, labels)

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

Concurrency in GUIs
- Be careful! In Swing, use SwingUtilities.invokeLater()