Event-Based Programming

Rob Miller
Fall 2008

Graphical User Interfaces

GUIs are composed from small reusable pieces

- button ( JButton )
- window ( JFrame )
- tree widget ( JTree )
- splitter bar ( JSplitPane )
- scrolling pane ( JSplitPane )

View Hierarchy

A GUI is structured as a hierarchy of views

- A view is an object that displays itself on a rectangular region of the screen

- Example: view hierarchy in GUIs
- Event-based programming
- Example: input handling in graphical user interfaces
- Model-view-controller pattern
  - Found throughout user interfaces

Today’s lecture

Composite pattern
- Example: view hierarchy in GUIs
Event-based programming
- Example: input handling in graphical user interfaces
Model-view-controller pattern
- Found throughout user interfaces
Composite Pattern

View hierarchy is an example of the Composite pattern
- Primitive views don’t contain other views
  - button, tree widget, text box, thumbnail, etc.
- Composite views are used for grouping or modifying other views
  - JScrollPane displays two views side-by-side with an adjustable splitter
  - JScrollPane displays only part of a view, with adjustable scrollbars

Key idea
- Primitives and composites implement a common interface (jComponent)
- Containers can hold any JComponent, so both primitives and other containers

How the View Hierarchy Is Used

Output
- GUIs change their output by mutating the view hierarchy
  - e.g., to show a new set of photos, the current Thumbnails are removed from the tree and a new set of Thumbnails is added in their place
- A redraw algorithm automatically redraws the affected views using the interpreter pattern (paint() method)

Input
- GUIs receive keyboard and mouse input by attaching listeners to views
  - (more on this in a bit)

Layout
- An automatic layout algorithm automatically calculates positions and sizes of views using the interpreter pattern (doLayout() method)
  - Specialized composites (JSplitPane, JScrollPane) do layout themselves
  - Generic composites (JPanel, JFrame) delegate layout decisions to a layout manager (e.g. FlowLayout, GridLayout, BorderLayout, ...)

Handling Mouse Input

Centralized approach?

```java
while (true) {
    read mouse click
    if (clicked on New Album) doNewAlbum();
    else if (clicked on Delete Album) doDeleteAlbum();
    else if (clicked on Add Photos) doAddPhotos();
    ...
    else if (clicked on an album in the tree) doSelectAlbum();
    else if (clicked on +/- button in the tree) doToggleTreeExpansion();
    ...
    else if (clicked on a thumbnail) doToggleThumbnailSelection();
    ...
}
```

Not modular!
- Mixes up responsibilities for button panel, album tree, and thumbnails all in one place

Input Handling on the View Hierarchy

Input handlers are associated with views
- Also called listeners, event handlers, subscribers, and observers

not to be confused with observer methods in a datatype
Event-Based Programming

**Control flow through a graphical user interface**
- A top-level event loop reads input from mouse and keyboard
- For each input event, it finds the right view in the hierarchy (by looking at the x,y position of the mouse) and sends the event to that view’s listeners
- Listener does its thing (e.g., modifying the view hierarchy) and returns immediately to the event loop

A Closer Look at Listeners

**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 a view’s bounds
- 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**
- JButton sends an action event when it is pressed (whether by the mouse or by keyboard)
- JTree sends a selection event when the selected element changes (whether by mouse or by keyboard)
- JTextBox sends change events when the text inside it changes for any reason

**Internet messaging**
- Email mailing lists
- IM chatrooms
Separating Frontend from Backend

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?

Model-View-Controller Pattern

Model-View-Controller (MVC) separates responsibilities
- View displays output
- Controller handles input
- Model stores application data

A More Detailed Look

Listener interface decouples the view from the controller (somewhat)

MVC with a Mutable Model

Controller mutates the model; model updates the view
Decoupling the Model from the View
More interfaces decouple the view and the model

Summary of MVC
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

Another MVC Example: Textbox

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
  • e.g. JTree view that displays the full filesystem tree, and a JLabel view that just displays the number of files
➢ Multiple views can simultaneously share the same model
➢ Views can be reused with other models, as long as the model implements an interface
  • e.g. JTree class (the view) and TreeModel 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
- Each vertical time line shows one object’s lifetime
- Horizontal arrows show calls and returns, trading control between objects
- Dark rectangles show when a method is active (i.e., has been called but hasn’t returned yet)

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 observer method calls may occur while the mutator is still in progress

Why is this a potential problem?

Pitfall #1: Specific Example

```java
class Filesystem {
    private Map<File, List<File>> cache;

    public List<File> getContents(File folder) {
        check for folder in cache, otherwise read it from disk and update cache
        for (File f: getContents(folder)) {
            f.delete();
            fireChangeEvent(f, REMOVED); // notify listeners that f was deleted
            cache.remove(folder); // cache is no longer valid for this folder
        }
    }

    public void deleteContents(File folder) {
        for (File f: getContents(folder)) {
            f.delete();
            fireChangeEvent(f, REMOVED); // notify listeners that f was deleted
            cache.remove(folder); // cache is no longer valid for this folder
        }
    }
}
```

Solution
- source must establish rep invariant before giving up control to any listeners
- often done simply by waiting to send events until end of mutator
Pitfall #2: Listener Calls Mutators

The listener might call mutator on the source
- e.g., when two sources are listening to each other in order to keep their state synchronized
- So calls to mutators may occur while mutator is still in progress

Why is this a potential problem?

Pitfall #2: Specific Example

Solution
- only send events when mutator actually causes a state change

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 mutator is still in progress

Why is this a potential problem?

Pitfall #3: Specific Example

class Source {
    private Listener[] listeners;
    private int size;
    public void removeListener(Listener l) {
        for (int i = 0; i < size; ++i) {
            if (listeners[i] == l) { listeners[i] = listeners[size-1]; --size; }
        }
        private void fireChangeEvent(...) {
            for (int i = 0; i < size; ++i) listeners[i].changed(...);   }
    }

    What happens if listeners[i] removes itself here?

    } Java collections (Set, List, etc) have the same problem:
    It's not safe to mutate a collection while you're iterating over it

Solution
- fire events by iterating over a copy of the listeners data structure
- or use javax.swing.EventListerList which copies only when necessary
Summary

View hierarchy
- Organizes the screen into a tree of nested rectangles
- 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