Event-Based Programming

Spring 2013
Today’s lecture

Design Patterns from a broader perspective

- Some examples

Problem Statement

- How to represent a GUI

Composite pattern

- Example: view hierarchy in GUls

Public subscribe pattern

- Example: listener

Event-based programming

- Example: input handling in graphical user interfaces

Model-view-controller pattern

- Found throughout user interfaces
What is a design pattern?

- **A standard solution to a common programming problem**
  - A design or implementation structure that achieves a particular purpose
  - A high-level programming idiom
  - Well understood by the practitioners
  - Easier to learn, manage, and support the application complexity

- **A technique for making code more flexible**
  - Reduce coupling among program components

- **Shorthand for describing program design**
  - A description of connections among program components
Example 1: Encapsulation (data hiding)

**Problem:** Exposed fields can be directly manipulated
- Violations of the representation invariant
- Dependences prevent changing the implementation

**Solution:** Hide some components
- Permit only stylized access to the object

**Disadvantages:**
- Interface may not (efficiently) provide all desired operations
- Indirection may reduce performance
Example 2: Subclassing (inheritance)

**Problem: Repetition in implementations**
- Similar abstractions have similar members (fields, methods)

**Solution: Inherit default members from a superclass**
- Select an implementation via run-time dispatching

**Disadvantages:**
- Code for a class is spread out, and thus less understandable
- Run-time dispatching introduces overhead
Example 3: Iteration

Problem: To access all members of a collection, must perform a specialized traversal for each data structure

- Introduces undesirable dependences
- Does not generalize to other collections

Solution:

- The implementation performs traversals, does bookkeeping
- Results are communicated to clients via a standard interface

Disadvantages:

- Iteration order is fixed by the implementation and not under the control of the client
Example 4: Exceptions

**Problem:**
- Errors in one part of the code should be handled elsewhere.
- Code should not be cluttered with error-handling code.
- Return values should not be preempted by error codes.

**Solution: Language structures for throwing and catching exceptions**

**Disadvantages:**
- Code may still be cluttered.
- It may be hard to know where an exception will be handled.
- Use of exceptions for normal control flow may be confusing and inefficient.
Example 5: Generics

Problem:
➢ Well-designed data structures hold one type of object

Solution:
➢ Programming language checks for errors in contents
➢ `List<Date>` instead of just `List`

Disadvantages:
➢ Slightly more verbose types
Example 6: Factories

**Problem:**
- Cannot only depend on the superclass or abstract class when an object has to be created.

**Solution:**
- Use a factory class for creating objects

**Disadvantages:**
- Slightly more verbose
Example 7: Visitors

**Problem:**
- Adding new functionality gets scattered throughout the subclasses

**Solution:**
- Use a visitor pattern to coalesce a new functionality in a single class

**Disadvantages:**
- Has to setup the subclasses to support the visitor pattern
- Complex control-flow that jumps between the visitor class and the subclasses
If the world was a sequential step-by-step process, programming it would be sweet
But we have to deal with a gooey mess

Many Components
- Video, images, buttons, widgets…
- Data and code of the program

On one display plane
- Multiple display components
- All need to coexist and even get nicely formatted

Taking on Input
- Mouse/keyboard activity everywhere
- Has to impact the relevant component

Reactive actions
- React to the user input
- Modify application state, update display

All in a nice object oriented design
- Decoupled, extensible

© Robert Miller 2009
Graphical User Interfaces

GUIs are composed from small reusable pieces

- button (JButton)
- window (JFrame)
- tree widget (JTree)
- splitter bar (JSplitPane)
- scrolling pane (JSplitPane)
A GUI is structured as a hierarchy of views

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

View hierarchy is an example of the Composite pattern

➢ Primitive views don’t contain other views
  • button, tree widget, textbox, thumbnail, etc.
➢ Composite views are used for grouping or modifying other views
  • JSplitPane 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 (actually an abstract class JComponent)
➢ containers can hold any JComponent, - both primitives and containers
How the View Hierarchy Is Used

Output

 GUls 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)

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, ...)

Input

 GUls receive keyboard and mouse input by attaching listeners to views
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**, and are part of Publish-Subscribe pattern
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
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

```java
class JComponent {
  ...
  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);
    }
  }
}
```

```java
interface MouseListener {
  void mousePressed(MouseEvent e);
  void mouseReleased(MouseEvent e);
  void mouseMoved(MouseEvent e);
  ...
}
```
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 the 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

What else?
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

![Diagram showing the Model-View-Controller pattern with classes]

- **View**
  - Thumbnail
  - thumbnailSelector

- **Controller**
  - ThumbnailSelector
  - toggle()

- **Model**
  - Photo
  - getFile()
A More Detailed Look

Listener interface decouples the view from the controller (somewhat)

- View
  - JComponent
  - Thumbnail
- Controller
  - ThumbnailSelector
- Model
  - Photo

MouseListener

Not completely decoupled – in practice, views and controllers often have to be tightly coupled.
MVC with a Mutable Model

Controller mutates the model; model updates the view

![Diagram showing MVC with a mutable model](image)

- **View**
  - FilesystemTree
  - observer methods (e.g., getRootFolder(), getFiles())

- **Controller**
  - PressedDelete (KeyListener)
  - keyPressed()

- **Model**
  - Filesystem
  - change events (e.g., fileDeleted())
  - mutator methods (e.g., deleteFile())
Another MVC Example: Textbox

 JT extField is a JComponent that can be added to a view hierarchy

 JT extField

 KeyListener is a listener for keyboard events

 KeyListener

 get text

 text change events

 move cursor

keypress events

 Document represents a mutable string of characters

 Document

 edit text

 How to decouple View from Model?
Decoupling the View from the Model

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 handles application state
- implements state-changing behavior
- sends change events to views
Model-View-Controller Advantages

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
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)

```java
interface Source {
    addListener()
    removeListener()
    getter()
    setter()
}

interface Listener {
    changeEvent()
}
```
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

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
    }
    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 RI before giving up control to listeners
➤ often done 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
- Calls to mutators may occur while mutator is still in progress

Why is this a potential problem?
Solution

➢ only send events when mutator actually causes a state change
➢ If JSlider is already at 45, it should not send a change event
Pitfall #3: Listener Removes Itself

A listener may call `removeListener()` on 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(...);
    }
}

Not safe to mutate a collection while you’re iterating over it
- Java collections have the same problem

Solution

- fire events by iterating over a copy of the listeners data structure
- or use javax.swing.EventListenerList which copies only when necessary
- or do removal from the list (or other list mutations) with an event
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

![Diagram showing concurrency in GUIs]

- **Main thread**
- **Event loop**
- **Swing event-handling thread**
- **Mouse**
- **Keyboard**
- **View hierarchy**
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() {
        content.add(thumbnail); ...
    }
});
```

The event loop handles one of these pseudo-events by calling run()
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