Using MVC with Swing Components

Georgia Tech

Jumping Ahead a Bit...

- We're going to cover a specific architectural approach to building UI components
- *Model-View-Controller*
- Classic architecture from Smalltalk 80
  - Model: data structures that represent the component's state
  - View: object responsible for drawing the component
  - Controller: object responsible for responding to user input

- Why talk about it now?
- Swing optionally allows a modified version of MVC as a way for building components
- I'd like you to use this approach for Homework #2

Some Swing History

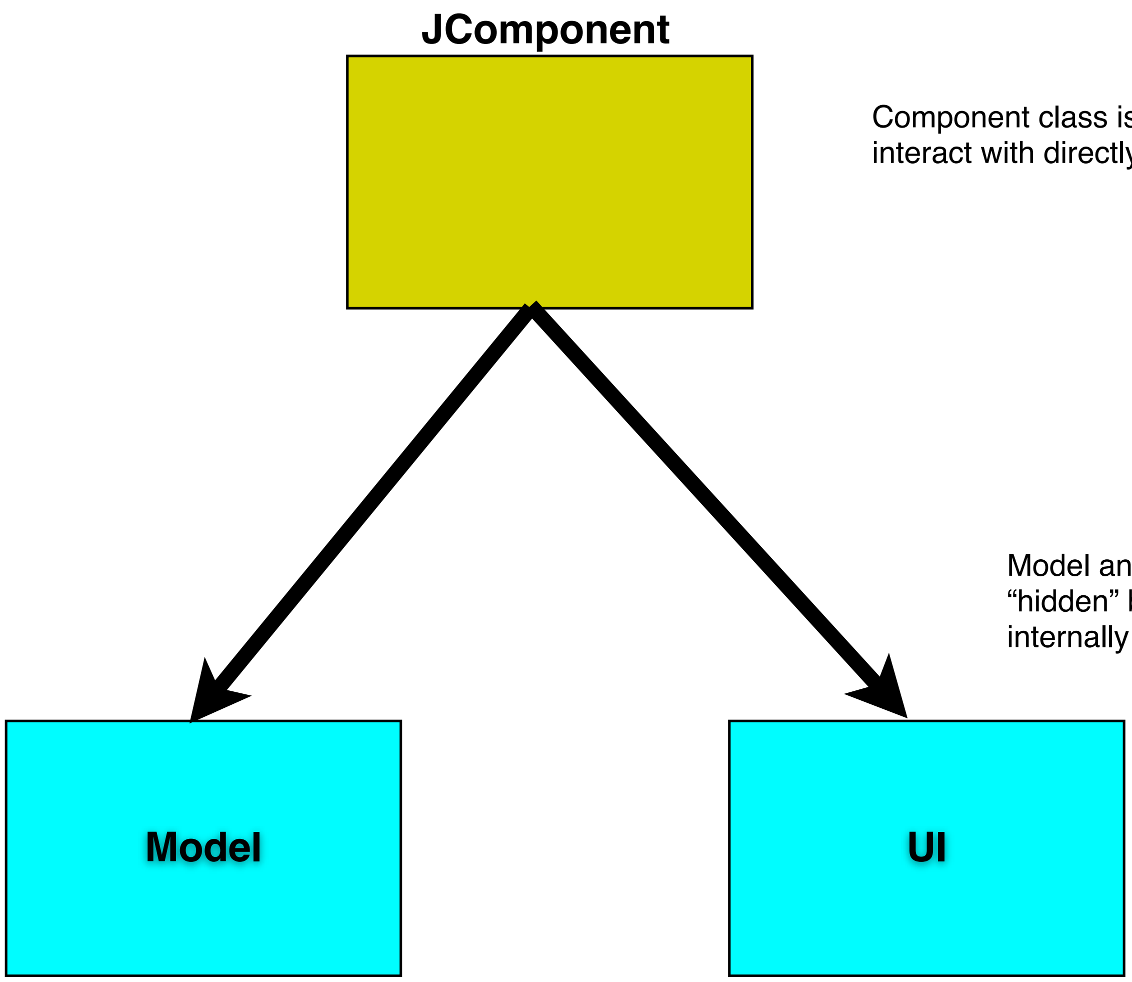
- Remember from earlier in class:
  - To create a new component, subclass JComponent
  - Implement paintComponent() to do all of the drawing for your component

- Nice, easy way to create components
- Still works fine
- But, makes some things very hard:
  - How would you implement a new look-and-feel?
  - Components' drawing code is hard coded into them.
  - Even if you had a big switch statement and implemented several look and feels, still doesn't help you if a *new* look and feel comes along.

Some Swing History (cont'd)

- Swing has a *pluggable look and feel* architecture (PLAF)
- Supports Windows, Mac, GTK, plus several Java-only LAFs
- To make these easier to use, many Swing components have factored their implementations in a slightly different way
  - Separation of the underlying component data from its look and behavior
- Allows you to create *just* a new look-and-feel for a component and easily plug it in to work with the core component data
    -

Component Internal Architecture



Component class is what applications typically interact with directly.

Model and UI classes are typically “hidden” by the component, and used internally by it.

Swing MVC Overview

- *Model:* custom class that contains all of the internal state of a component
- *UI:* custom class that handles user input events, and painting the component
  - Subsumes both the View and Controller from the classic MVC architecture
- These two classes are *loosely-coupled*
  - They communicate with each other through events
  - E.g., when something in the model updates, it sends a ChangeEvent to whatever UI is associated with it.
  - UI then calls repaint() to tell the RepaintManager to schedule it for redrawing.

Swing MVC Overview

- Application programmers typically never see the UI or the Model classes
  - Used purely as an internal implementation feature of the component
- Requires a bit of structure and boilerplate code to make things work right.
- Resources:
  - Short overview article: *MVC Meets Swing*, linked off class website
  - Book: last chapter covers creating new Swing components using this architecture

Step 1: Create Your Model Class

- Model: responsible for storing the state of your component
- Reuse an existing model if one is suitable; create your own if not
- 1. Create an interface for your model and an implementation class, if you're defining a new one
    - Decide on the data structures you'll need to track, and create getter/setter functions
        - Called *Properties* if they match the standard Java-style standards
- 2. Send PropertyChangeEvents (or just ChangeEvents) when data in the model change
    - This means you'll need to keep a list of PropertyChangeListeners (or just ChangeListeners), and provide methods for adding and removing listeners
    - EventListenerList can help with this
- Be careful: the model should *only* contain core data structures, *not* data that's only about the visual presentation of that data
    - Example: a Scrollbar
    - Minimum, maximum, and current values are model properties (they have to do with actual data values, not display
    - Whether tick marks are shown, labels, etc., are visual properties, and don't belong in the model (they're only about display, not the actual data)

Step 2: Create an Abstract UI Class

- This is an abstract superclass to be shared by all LaFs for your new component
- Will be the superclass of all UIs that are "compatible" with your new component (for this phase of the project, there will be only one class that subclasses it)
- Always follows the same basic format:

```
import javax.swing.plaf.ComponentUI;

public abstract class NotepageUI extends ComponentUI {
        public static final String UI_CLASS_ID = “NotepageUI”;
}
```

Step 3: Create the Actual UI Class

- 1. Extend your abstract UI class
- 2. Implement public void paint(Graphics g, JComponent c)
  - Your component will automatically delegate its drawing to your UI's paint() method
- 3. Implement any interfaces you need in order to respond to input events
  - Example: if your component must respond to the mouse, have your UI class implement MouseListener. You'll tell the component to send any mouse events to your UI to be handled there.
- 4. Draw yourself correctly given your current size
  - Recall that your parent component may resize you! In your painting code, use the current size (getWidth()/getHeight()) and draw in the space alloted to you.
- 5. Implement a bit of boilerplate code for UI management
  - public static ComponentUI createUI(JComponent c);
    - Create and return an instance of your UI class
  - public void installUI(JComponent c);
    - Register 'this' UI instance as the listener for the component's input events
  - public void uninstallUI(JComponent c);
    - Unregister 'this' UI instance as the listener for the component's input events

Step 4: Create the Component Itself

- 1. Design the component's external API
  - These are the methods that application programmers see and use
  - Many will just forward to the underlying model or the UI
- 2. Make your component a listener for the Model's ChangeEvents or PropertyChangeEvents
  - Generally need to call repaint() whenever the model is updated
- 3. Send PropertyChangeEvents if *the component's* internal state changes
  - Other components might be listening to you--send PropertyChangeEvents if anything component-specific changes
- 4. Implement some boilerplate methods to register models and UIs
  - public void setUI();
  - public void updateUI();
    - Used to set the UI, and change it on the fly
  - public String getUIClassID();
    - Should return whatever the UI_CLASS_ID string is for "compatible" UIs for this component
  - public void setModel();
  - public Model getModel();
    - Used to set and return the model. When the your model is set, your component should register itself as a listener for the model's change events.

Step 5: Register your UI with Swing’s UIManager

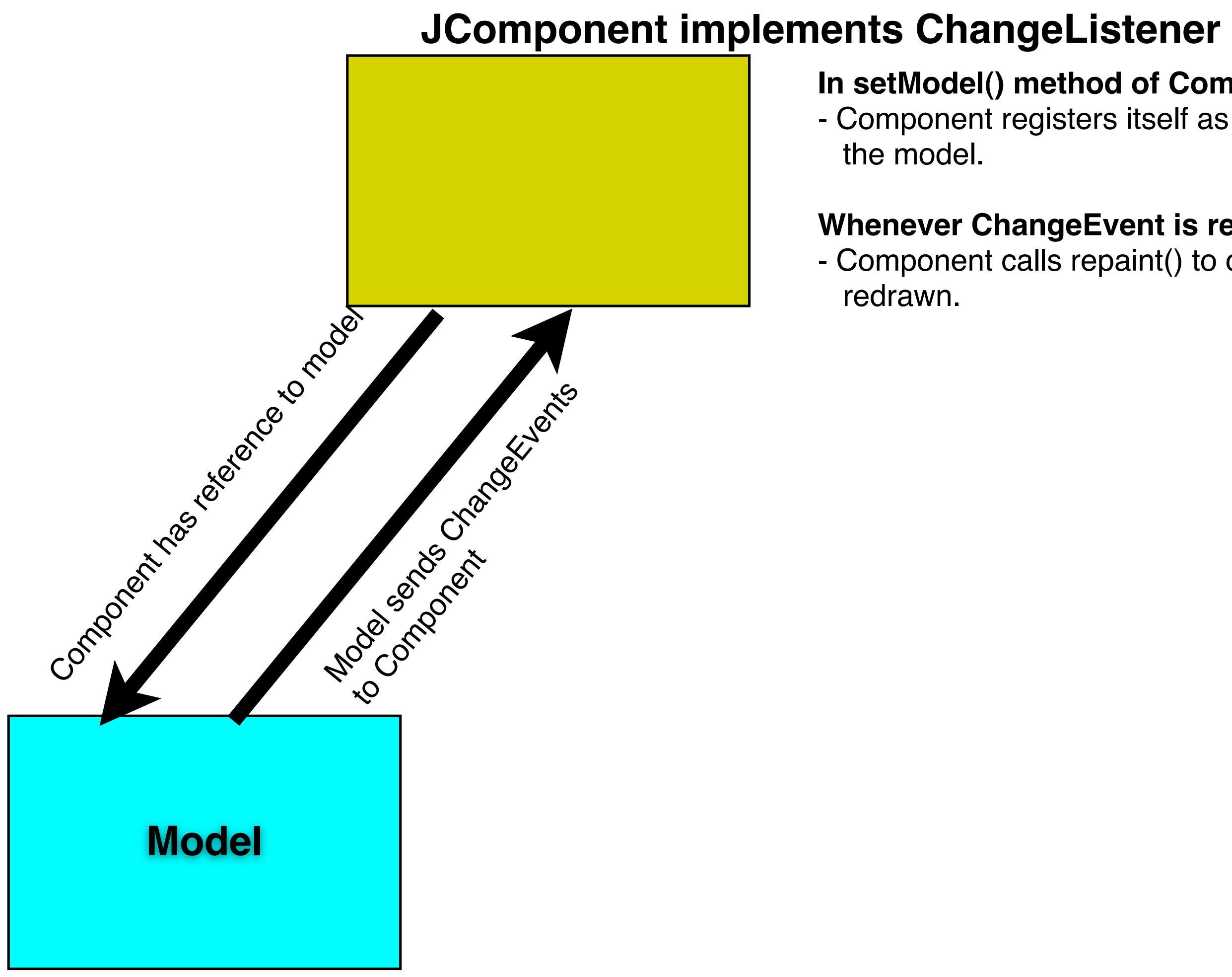
- Need to tell the UIManager about the specific UI you want to use
- Typically do this early in the application’s main() routine:

```
public static void main(String[] args) {
    UIManager.put(PhotoUI.UI_CLASS_ID, “BasicNotepageUI”);
    // ... other stuff here ...
}
```

This string serves as the unique token identifying all different UIs that work as NotepageUIs

This string names the class that implements the specific look-and-feel UI you want to use in this application

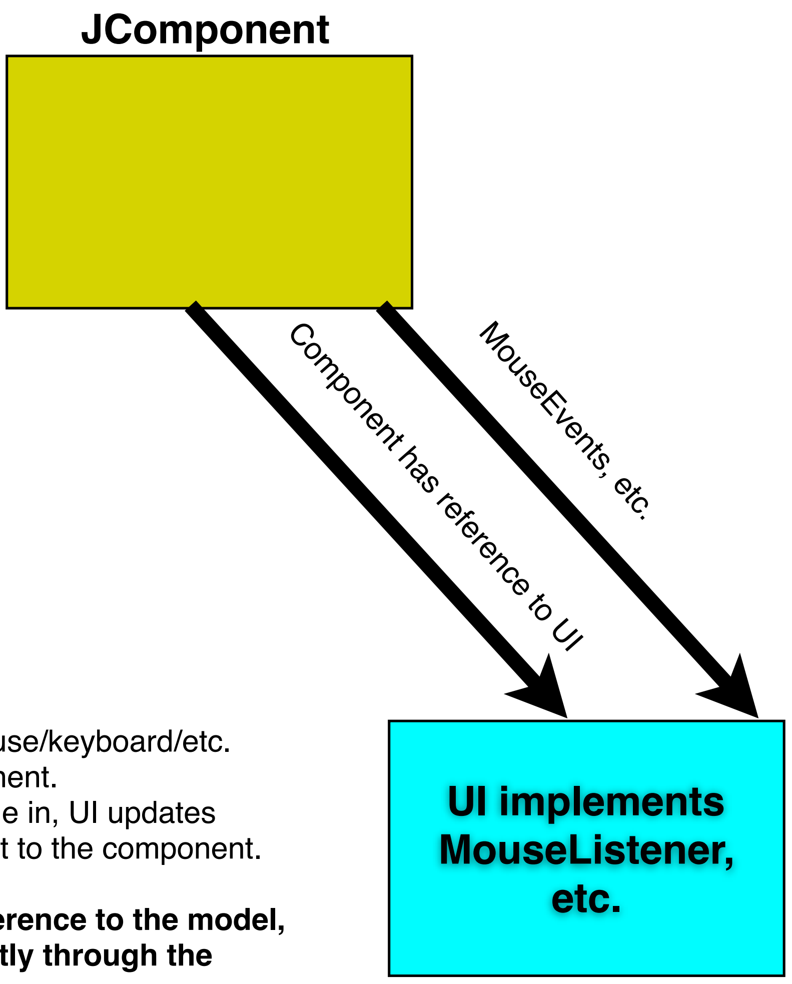
Component Internal Architecture


JComponent implements ChangeListener
Component has reference to model
Model sends ChangeEvents to Component
Model

In setModel() method of Component:
- Component registers itself as a ChangeListener for the model.

Whenever ChangeEvent is received from model:
- Component calls repaint() to cause itself to be redrawn.

Component Internal Architecture


JComponent
Component has reference to UI
MouseEvents, etc.
UI implements MouseListener, etc.

In installUI() method:
- UI sets itself up as mouse/keyboard/etc. listener for the component.
- When user events come in, UI updates the model by calling out to the component.

UI does *not* have a reference to the model, but accesses it indirectly through the Component.

In paint() method:
- Component is passed in to paint()
- Ask component for data that needs to be drawn

UI does *not* have a reference to the model, but accesses it indirectly through the Component

Step 3 (example)

```
public class BasicNotepageUI extends NotepageUI implements MouseListener {
    public static ComponentUI createUI(JComponent c) {
        return new BasicNotepageUI();
    }
    public void installUI(JComponent c) {
        ((NotepageComponent) c).addMouseListener(this); // we'll handle mouse events for the Notepage component
    }
    public void uninstallUI(JComponent c) {
        ((NotepageComponent) c).removeMouseListener(this);
    }
    public void paint(Graphics g, JComponent c) {
        // do painting for the component here!
    }

    // implement the various MouseListener methods...
}
```

Step 4 (Example)

```
public class NotepageComponent extends JComponent implements ChangeListener {
    NotepageModel model;
    public NotepageComponent() {
        setModel(new NotepageModel());
        updateUI();
    }
    public setModel(NotepageModel m) {
        if (model != null)
            model.removeChangeListener(this);
        model = m;
        model.addChangeListener(this);
    }
    public NotepageModel getModel() {
        return model;
    }
    public void setUI(NotepageUI ui) { super.setUI(ui); }
    public void updateUI() {
        setUI((NotepageUI) UIManager.getUI(this));
        invalidate();
    }
    public String getUIClassID() { return NotepageUI.UI_CLASS_ID; }
}
```

Common Problems

- Exceptions at startup time
  - Make sure the UIManager registration is done before you use the component
- Components aren't being repainted all the time
  - Make sure you're registered for change events, and are calling repaint() whenever anything changes
- Components come up at weird sizes
  - Your component should provide a miminumSize and preferredSize when it is requested. If you don't do this, your parent may set your size to 0