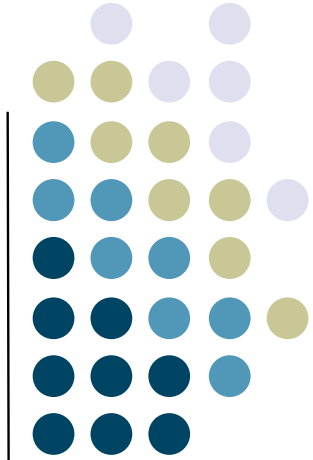
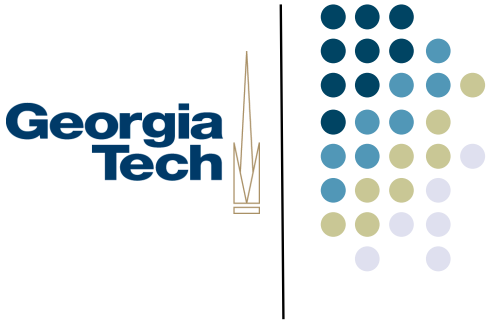
Using MVC with Swing Components



Georgia Tech

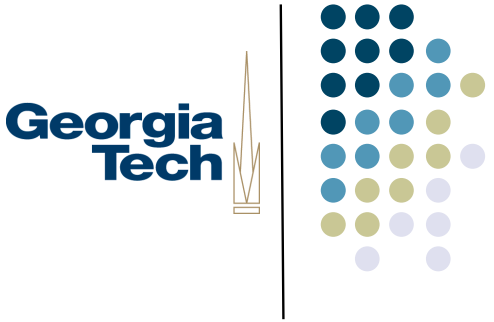


Jumping Ahead a Bit...

- 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

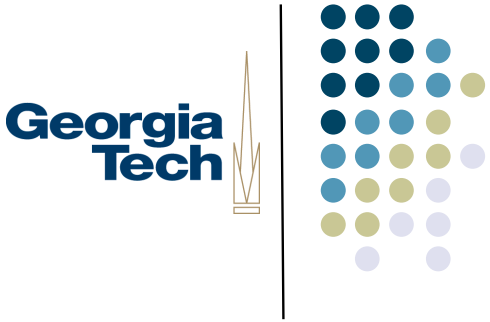
- Why talk about it now?
- Swing optionally allows a modified version of MVC as a way for building components
- I'd like you to use this approach for Homework #2

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

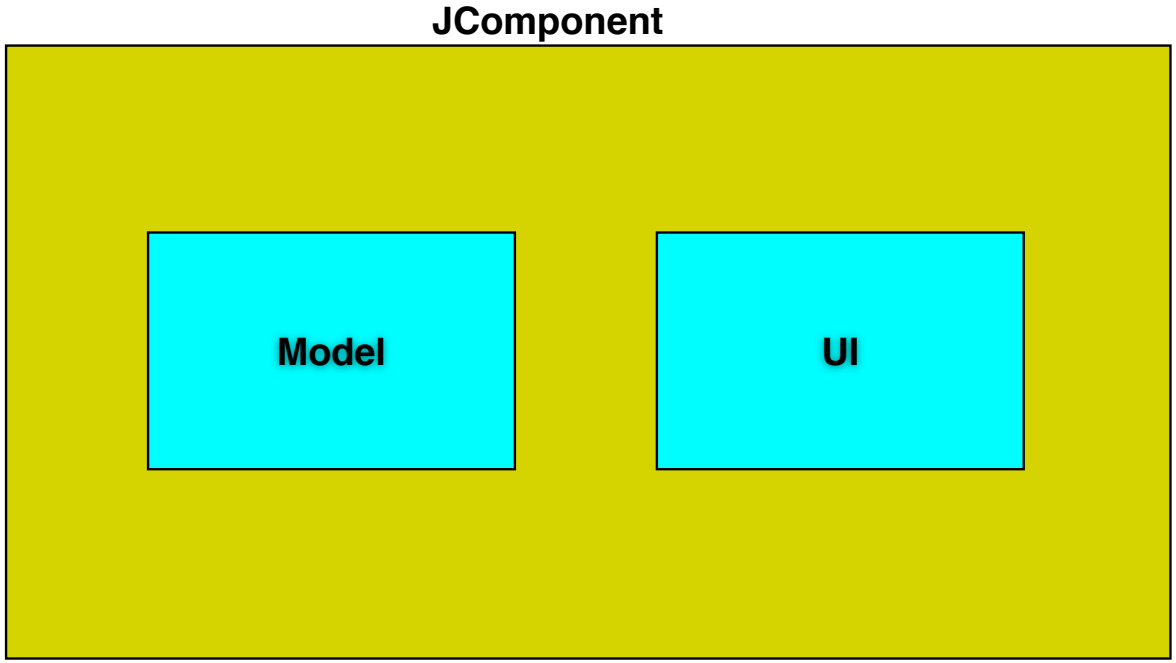
- Nice, easy way to create components
- Still works fine
- But, makes some things very hard:
  - How would you implement a new look-and-feel?
  - Components' drawing code is hard coded into them.
  - Even if you had a big switch statement and implemented several look and feels, still doesn't help you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



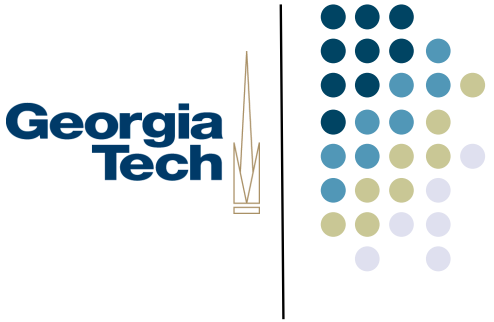
Some Swing History (cont'd)

- 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

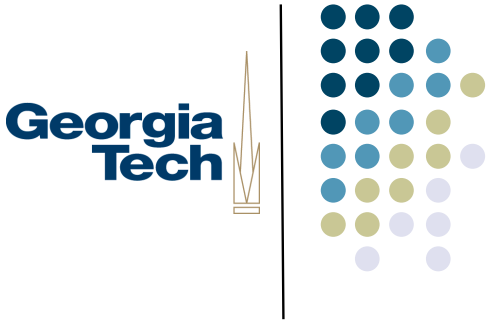


Swing MVC Overview



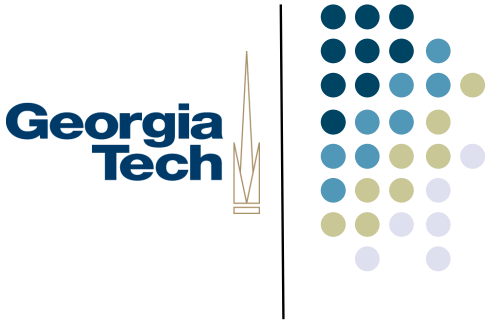
- *Model:* custom class that contains all of the internal state of a component
- *UI:* custom class that handles user input events, and painting the component
  - Subsumes both the View and Controller from the classic MVC architecture
- These two classes are *loosely-coupled*
  - They communicate with 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.

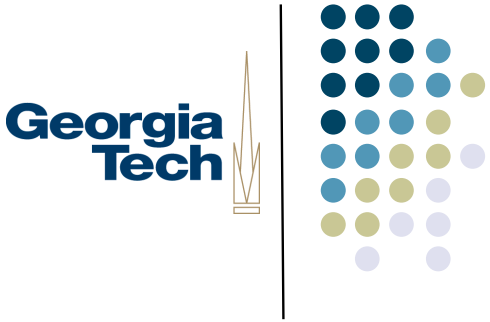
- Resources:
  - Short overview article: *MVC Meets Swing*, linked off class website
  - Book: last chapter covers creating new Swing components using this architecture



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
- 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
- Send PropertyChangeEvents (or just ChangeEvents) when data in the model change
- Keep a list of PropertyChangeListeners (or just ChangeListeners), and provide methods for adding and removing listeners
- 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)

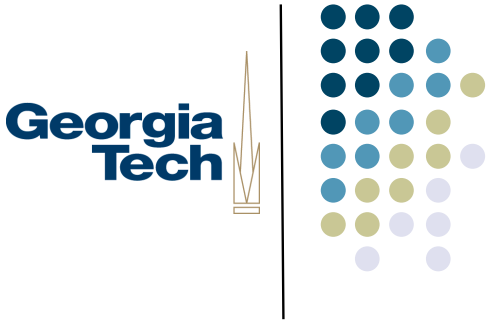
Step 2: Create an Abstract UI Class



- This is an abstract superclass to be shared by all LaFs for your new component
- Always follows the same basic format:

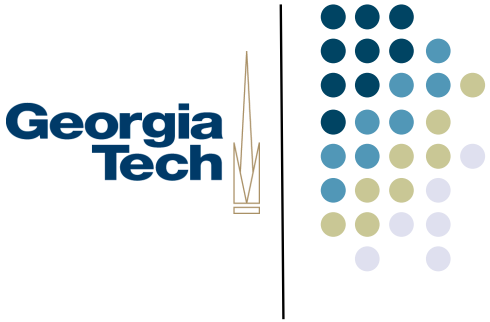
```
import javax.swing.plaf.ComponentUI;

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

Step 3: Create the Actual UI Class



- Extend the abstract UI class
- Implement public void paint(Graphics g, JComponent c)
  - Your component will automatically delegate its drawing to your UI's paint() method
- 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
- 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.
- Implement a bit of boilerplate code for UI management

Step 4: Create the Component Itself

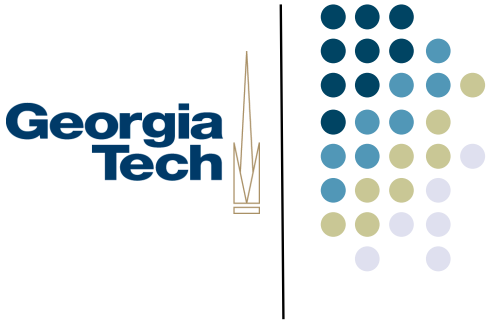


- Design the component's external API
  - These are the methods that application programmers see and use
  - Many will just forward to the underlying model or the UI
- Make your component a listener for the Model's ChangeEvents or PropertyChangeEvents
  - Generally need to call repaint() whenever the model is updated
- Send PropertyChangeEvents if *the component's* internal state changes
  - Other components might be listening to you--send PropertyChangeEvents if anything component-specific changes
- Implement some boilerplate methods to register models and UIs

Step 4 (Example)

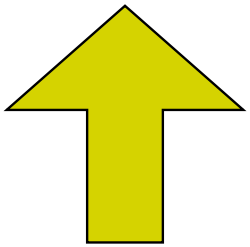
```
public class Notepage extends JComponent implements ChangeListener {
    NotepageModel model;
    public Notepage() {
        setModel(new NotepageModel());
        updateUI();
    }
    public setModel(NotepageModel m) {
        old = this.model;
        if (old != null) old.removeChangeListener(this);
        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; }
}
```

Step 5: Register your UI with Swing's UIManager

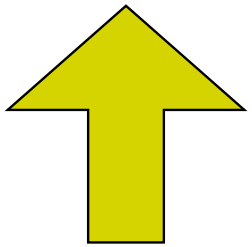


- Need to tell the UIManager about the specific UI you want to use
- Typically do this early in the application's main() routine:

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

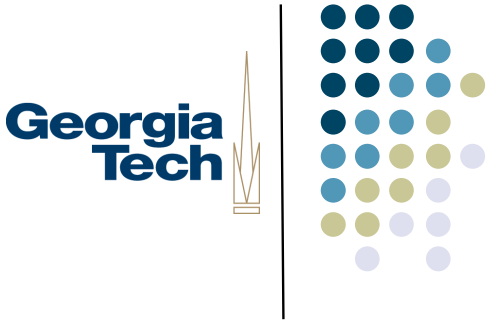


This string serves as the unique token identifying all different UIs that work as NotepageUIs



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

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

Step 3 (example)

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

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