The user interface

- From previous class:
  - Generally want to think of the “UI” as only one component of the system
    - Deals with the user
    - Separate from the “functional core” (AKA, the “app”)
Separation of Concerns

- There are good software engineering reasons to do this
  - Keep UI code separate from app code
  - Isolate changes
  - More modular implementation
  - Different expertise needed
  - Don’t want to iterate the whole thing
In practice, very hard to do...

- More and more interactive programs are tightly coupled to the UI
  - Programs structured around UI concepts/flow
  - UI structure “sneaks into” application
- Not always bad...
  - Tight coupling can offer better feedback/performance
Separation of concerns is a central theme of UI organization

- A continual challenge
- A continual tension and tradeoff
- Real separation of UI from application is almost a lost cause
Conceptual Overview of the UI

UI Toolkit

UI Core

Input

Output

App Interface

App
Basic UI Flow

Input

Output

UI Core

App

Interface

UI Toolkit

App
How would you architect this?

- Tempting to architect systems around these boxes
  - One module for input, one for output, etc.
  - Has been tried ("Seeheim model")
  - Didn’t work well
Why “Big Box” architectures don’t work well

- Modern (“direct manipulation”) interfaces tend to be collections of quasi-independent agents
  - Each interactor (“object of interest” on the screen) is separable
  - Example: an on-screen button
    - Produces “button-like” output
    - Acts on input in a “button-like” way
    - Etc.
Leads to object-based architectures

• Each on-screen interactor corresponds to an object instance
  • Common methods for
    • Drawing output (button-like appearance)
    • Handling input (what happens when I click)

• Objects are organized hierarchically at runtime
  • Normally reflecting spatial containment relationships
  • NOTE: different than class hierarchy created at development time

• Interactor trees
Challenge: maintaining separation of concerns

- Trick is coming up with a separation that works quickly, simply, and extensibly
  - Even a single button may be hopelessly complex (pluggable looks-and-feels anyone?)
  - Needs to be extensible to new interactors
  - What’s the right factoring for all this stuff?
- Will see some strategies later
- Basically: common O-O patterns to manage complexity
UI Toolkits

- System to provide development-time and runtime support for UIs
  - Core functionality
  - Input & output handling
  - Connecting to the application
- Also: specific interaction techniques
  - Library of interactors
  - Look and feel (sometimes pluggable)
Categories of users

- Consumer
  - End-user, albeit indirectly

- Programmers
  - Interface designer
  - Application builder
  - Toolkit implementer/maintainer
  - Interactor writer
  - Tool builder
  - Expert end-user (through scripting)
Toolkit functionality in detail (Roadmap of topics)

- Core functions
  - Hierarchy management
    - Create, maintain, tear down tree of interactor objects
  - Geometry management
    - Dealing with coordinate systems
    - On-screen bounds of interactors
  - Interactor status/information management
    - Is this interactor visible? Is it active?
Toolkit functionality in detail

- Output
  - Layout
    - Establishing the size and position of each object
    - Both initially, and after a resize
  - (Re)drawing
  - Damage management
    - Knowing what needs to be redrawn
  - Localization and customization
    - We won’t talk much about this...
Toolkit functionality in detail

- Input
  - Picking
    - Figuring out what interactors are “under” a given screen point
  - Event dispatch, translation, handling
    - This is where a lot of the work goes
Toolkit functionality in detail

- Application interface
  - How the UI system connects with application code
    - Callbacks
    - Command objects
    - Undo models
    - ...

Example: Java Swing

- All functions of interactors encapsulated in base class
  - javax.swing.JComponent
  - All objects on-screen inherit from this class

- Terminology:
  - interactor, widget, component, control, ...
Standard object-oriented approach

- Base class (or interface) defines the set of things that every interactor must do
  - e.g., public void paintComponent(Graphics g);
- Subclasses provide specific specialized implementations
  - Do the right drawing, input, etc., to be a button vs. a slider vs. ...
JComponent API defines methods for

- Hierarchy management
- Geometry management
- Object status management
- Layout
- (Re)drawing
- Damage management
- Picking
In subclasses and other parts of the toolkit:

- Input dispatch and handling
- Application interface
- Pluggable looks and feels
- Undo support
- Accessibility
Hierarchy Management

- Swing interfaces are trees of components
- To make something appear, you must add it to the tree
- Swing takes care of many of the details from there
  - Screen redraw
  - Input dispatch
Hierarchy Management

- Lots of methods for manipulating the tree
  - add(), remove(), removeAll(), getComponents(), getComponentCount(), isAncestorOf(), ...
- Common mistake
  - If nothing shows up on the screen, make sure you’ve added it!
Geometry Management

- Every component maintains its own geometry:
  - Bounding box: getX(), getY(), getWidth(), getHeight()
    - X,Y are relative to parent
    - i.e., 0,0 is at parent’s top left corner
    - Other operations: setSize(), setLocation(), setBounds(), getSize(), getLocation(), getBounds()
  - All drawing happens within that box
    - System clips to bounding box
    - Including output of children!
  - Drawing is relative to top-left corner
    - Each component has its own coordinate system
Object Status

- Each component maintains information about its “state”
  - isEnabled(), setEnabled()
  - isVisible(), setVisible()
- Lots of other methods of lesser importance
Each component handles:

- Layout (we’ll talk about this later...)
- Drawing
  - Each component knows how to (re)create its appearance based on its current state
  - Responsible for painting three items, in order:
    1. Component
    2. Borders
    3. Children
  - paintComponent(), paintBorder(), paintChildren()
  - **These are the only places to draw on the screen!!!**
- Automatically called by JComponent’s paint() method, which is itself called by the Swing RepaintManager (figures out “damaged” regions)
Damage Management

- **Damage:** areas of a component that need to be redrawn
  - Sometimes: computed automatically by Swing RepaintManager
    - e.g., if another window is dragged over your component, or your component is resized
  - Other times: you need to flag damage yourself to tell the system that something in your internal state has changes and your on-screen image may not be correct
    - e.g., your component needs to change the color of a displayed label

- **Managing damage yourself:**
  - `repaint(Rectangle r)`
  - Puts the indicated rectangle on the RepaintManager’s queue of regions to be redrawn

- **Terminology:** `damage` is not a Swing term; generic
Picking

- Determine if a point is “inside” a component
  - `contains(int x, int y)`
  - Is the point inside the bounding box of this component (uses local coordinate system of component)
- Terminology: likewise, *picking* is not a Swing term
Other stuff

- Input (we’ll talk about this later...)
- Application interface
  - Glue between component and application functionality
  - Not directly in component, but there is a convention for how to associate your functionality with a component
  - Callbacks: you register code with a component to say “call this code when something happens”
- Terminology: Swing uses the term listener for a piece of application code that will be called back in response to something happening
  - The code “listens for” something happening
Listeners

- Any given component may have multiple situations in which it invokes a listener
  - Button pressed, list scrolled, list item selected
  - Different types of listeners representing different types of things happening
- Therefore, each component has a list of listeners for each situation
- Standardized names for accessing these lists
  - addPropertyChangeListener(), getPropertyChangeListener(), removePropertyChangeListener()
  - addActionListener(), getActionListeners(), removeActionListener()
More on listeners

- There is generally a separate interface for each type of listener
  - PropertyChangeListener
  - ActionListener

- Your code must implement the appropriate listener interface and be registered with the list of appropriate list of listeners on the appropriate component
  - Example: button press causes listeners on the button’s ActionListener list to be called
    - Define your code so that it implements ActionListener
    - Register it with the button using addActionListener()
Events

- Most listener interfaces define methods that take an *event object* that describes what just happened
- Separate classes of events for each listener interface
  - ActionListener: ActionEvent
  - MouseListener: MouseEvent
- Passed as a parameter containing details of what happened
  - e.g., MouseListener: mouse coordinates, whether it was pressed, released, etc.