javax.swing, events, and GUIs

- GUI programming requires processing events
  - There’s no visible loop in the program
  - Wire up/connect widgets, some generate events, some process events
    - Pressing this button causes the following to happen
  - We want to practice “safe wiring”, meaning?

- We need to put widgets together, what makes a good user-interface? What makes a bad user-interface?
  - How do we lay widgets out, by hand? Using a GUI-builder? Using a layout manager?

- How do we cope with widget complexity?

Wiring Buttons and Events and GUIs

- Button/Menu generates event
  - ActionEvent
- Click mouse generates event
  - MouseEvent
- When creating widget/button
  - Attach listeners for events
  - Can have multiple listeners
- Activating widget/button
  - Causes listeners to “hear”
  - Listeners act accordingly
- Press E, handle E-event
  - What happens?
- Press A, handle A-event ...

Action and other events

- Widgets generate events, these events are processed by event listeners
  - Different types of events for different scenarios: press button, release button, drag mouse, press mouse button, release mouse button, edit text in field, check radio button, ...
  - Some widgets “fire” events, some widgets “listen” for events
- To process events, add a listener to the widget, when the widget changes, or fires, its listeners are automatically notified.
  - Firing widget is the “model”, listener is the “view” aka observable and observer, respectively

Code to wire widgets/events/listeners

- Old-fashioned way, make the GUI the listener – has doNext ()
  - Listener is an object, with the right method (kind of event)

```java
myNextButton = new JButton("next");
myNextButton.addActionListener(this);
public void actionPerformed(ActionEvent e) {
    if (e.getSource() == myNextButton) doNext();
}
```
- New way, create anonymous event-listener object/on-the-fly
  - Inner class, has access to/can call GUI functions

```java
myNextButton.addActionListener(
    new ActionListener(){
        public void actionPerformed(ActionEvent e) {
            doNext();
        }
    });
```
Adding Listeners

- In lots of code you’ll see that the Container widget is the listener, so pressing a button or selecting a menu is processed by the Container/Frame’s actionPerformed method
  - All ActionListeners have an actionPerformed method, is this interface/implements or inheritance/extends?
  - Here’s some “typical” code, why is this bad?

```java
void actionPerformed(ActionEvent e)
{
  if (e.getSource() == thisButton) …
  else if (e.getSource() == thatMenu)…
}
```

A GUI object can be its own client

- Occasionally a GUI will be a listener of events it generates
  - Simple, but not extendable
  - Inner classes can be the listeners, arguably the GUI is still listening to itself, but ...
    - Encapsulating the listeners in separate classes is better

- Client (nonGUI) objects cannot access GUI components
  - Properly encapsulated JTextField, for example, responds to aGui.displayText()
  - Tension: simplicity vs. generality

- Don’t wire widgets together directly, use methods/classes
  - Eventual trouble when GUI changes

Using inner/anonymous classes

- For each action/event that must be processed, create an object to do the processing
  - Command pattern: parameterize object by an action to perform, queue up requests to be executed at different times, support undo
  - There is a javax.swing Event Queue for processing events, this is the hidden while loop in event processing GUI programs

- The inner class can be named, or it can be created “anonymously”
  - For reuse in other contexts, sometimes naming helpful
  - Anonymous classes created close to use, easy to read (arguable to some)

Listeners

- Events propagate in a Java GUI as part of the event thread
  - Don’t manipulate GUI components directly, use the event thread
  - Listeners/widgets register themselves as interested in particular events
    - Events go only to registered listeners, can be forwarded/consumed

- ActionListener, KeyListener, ItemListener, MouseListener, MouseMotionListener, …, see java.awt.event.*

- Isolate listeners as separate classes, mediators between GUI, Controller, Application
- Anonymous classes can help here too
**JComponent/Container/Component**

- The `java.awt` package was the original widget package, it persists as parent package/classes of `javax.swing` widgets
  - Most widgets are `JComponents` (subclasses), to be used they must be placed in a `Container`
    - The former is a swing widget, the latter `awt` `J` convention.
- A `Container` is also a `Component`
  - `JFrame` is often the “big container” that holds all the GUI widgets, we’ll use this and `JApplet` (awt counterparts are `Frame` and `Applet`)
  - A `JPanel` is a `JComponent` that is also a `Container`
    - Holds `JComponents`, for example and is holdable as well

**Creating a GUI**

- Create Frame (applet)
  - `Application/Applet`
- Create main panel
  - `BorderLayout`
- Create button panels
  - `Top/Bottom, wired`
- Create menus
  - Wired when created
- Create main view/panel
  - Shows what’s happening
- Create non-view logic and attach appropriately
  - Debug 5 days and serve

**Creating a GUI (again)**

- Create Frame (applet)
  - `Application/Applet`
- Create main panel
  - `BorderLayout`
- Create button panels
  - `Top/Bottom, wired`
- Create menus
  - Wired when created
- Create main view/panel
  - Shows what’s happening
- Create non-view logic and attach appropriately
  - Debug 5 days and serve

**Widget layout**

- A Layout Manager “decides” how widgets are arranged in a `Container`
  - In a `JFrame`, we use the `ContentPane` for holding widgets/components, not the `JFrame` itself
  - Strategy pattern: “related classes only differ in behavior, configure a class with different behaviors… you need variants of an algorithm reflecting different constraints…context forwards requests to strategy, clients create strategy for the context”
    - Context == `JFrame/container`, Strategy == `Layout`
- Layouts: `Border, Flow, Grid, GridBag, Spring, ...`
  - I’ll use `Border, Flow, Grid` in my code
BorderLayout (see Browser.java)

- Default for the JFrame content pane
- Provides four areas, center is "main area" for resizing
- Recursively nest for building complex (yet simple) GUIs
- BorderLayout.CENTER for adding components
  - Some code uses "center", bad idea (bugs?)

JFrame/JApplet things to remember

- setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
  - Sometimes see WindowListener, previous JDK
  - Not used in applets
- getContentPane(panelWithEverything);
  - Create panel, add to top-level content pane
- setVisible(true);
  - Must have this in application
- JFrame
  - setSize(x,y); pack(); setTitle(..)
- JApplet
  - init(); arguments via getParameter(..)

Model, View, Controller overview

- Model, View, Controller is MVC
  - Model stores and updates state of application
    - Example: calculator, what's the state of a GUI-calculator?
  - When model changes it notifies its views
    - Example: pressing a button on calculator, what happens?
  - The controller interprets commands, e.g., button-click, forwards them appropriately to model (not to view)
    - Example: code for calculator that reacts to button presses
  - Controller isn't always a separate class, often part of GUI-based view in M/VC
- MVC is a fundamental design pattern: solution to a problem at a general level, not specific code per se

MVC in SameGame/Clickomania

- Clicking on a piece connected to more than one causes pieces to be removed
  - See GameGui, GameCore, GameModel, GameApplet, GameMove
- The model "knows" the location of the pieces
  - Determines if move is legal
  - Makes a move (records it) and updates views
- View shows a board and information, e.g., undo possible
  - See IGameView interface, implemented by application and applet via GameCore
**SameGame MVC: Controller**

- Here the GameCore communicates with model so acts as controller together with ButtonPanel
  - Sometimes a middleman class isn’t a good idea, extra layer of code that might not be needed
  - Often in MVC model communicates with multiple views, but communication to model via controller

- GameCore implements IGameView, easy to incorporate new controllers/views
  - Model doesn’t know about implementation
  - Model doesn’t know about button panels

**Controller and Commands**

- **Use-case for making a move:**
  - Move generated, e.g., by button-click
  - Move forwarded to model (by controller)
    - (If move is made, then undo state set to true in views)
    - If move is made, then views update board display

- **Use-case for undoing a move**
  - Undo generated, e.g., by button or menu choice
  -Undo forwarded to model (by controller)
    - If undo changes board, views will be updated
    - If future undo not possible, undo state set to false