From Using to Programming GUIs

- Extend model of "keep it simple" in code to GUI
  - Bells and whistles ok, should be easy to use and hide

- We're talking about software design
  - Not HCI or user-interface design or human factors…
  - However, compare winamp to iTunes

- How do we design GUIs
  - Programming, drag-and-drop, …
  - How do we program/connect GUIs?

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 do 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?

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, why?

- A Container is also a Component, but not all Containers are JComponents (what?)
  - ➢ 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

What do Containers do?

- A Container is a Component, so it's possible for a Container to "hold itself"? Where have we seen this?

"You want to represent part-whole hierarchies of objects. You want clients to be able to ignore the difference between compositions of objects and individual objects. Clients will treat all objects in the composite structure uniformly".

- Composite pattern solves the problem. Think tree, linked list: leaf, composite, component
  - What about parent references?
  - What about child references?
- In java, a parent is responsible for painting its children
  - For "paint" think draw, arrange, manage, …

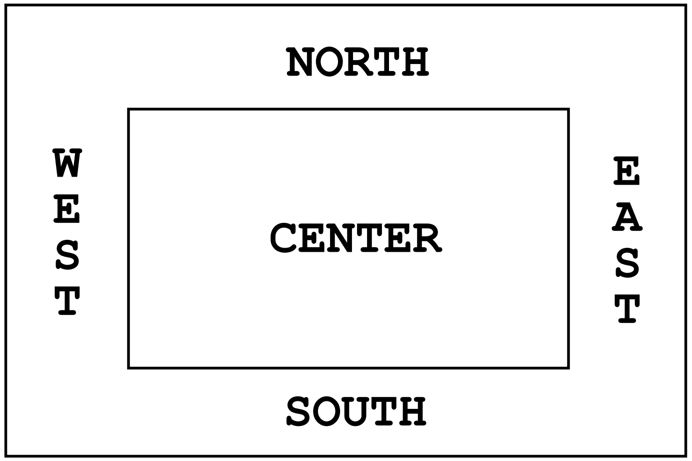
Widget layout

- 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 contentpane
- 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?)



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.
  - Observer/Observable (related to MVC) pattern

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?

```
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(), textfield not accessible to clients
  - If the GUI is its own client, it shouldn't access textfield
    - Tension: simplicity vs. generality
- Don't wire widgets together directly or via controller that manipulates widgets directly
  - 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

Listeners and Adapters

- MouseListener has five methods, KeyListener has three
  - What if we're only interested in one, e.g., key pressed or mouse pressed?
    - As interface, we must implement all methods as no-ops
    - As adapter we need only implement what we want

- Single inheritance can be an annoyance in this situation
  - Can only extend one class, be one adapter, …

- What about click/key modifiers, e.g., shift/control/left/both
  - Platform independent, what about Mac?

From Browser/Memory to OOGA

- Can you design a game architecture before designing a game?
  - Which games should you write, do you need to have full-blown implementations?
  - What's enough to start?

- Draw (on paper) a picture of the GUI
  - Generate scenarios, use-cases: what happens when user clicks here, what happens when user enters text in this box, what happens when game is over
  - Rough-out the components, prototype a GUI
  - Connect some/all components, make it possible to refactor

- It's hard to make a framework without making a frame

MVC 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, forwards them appropriately to model (usually 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 in the jpuzzle suite

- Sliding puzzle game allows users to move pieces near a blank piece to recreate an original image
  - See PuzzleGui, PuzzleController, PuzzleView, PuzzleModel, PuzzleApplet, PuzzleMove

- The model "knows" the location of the pieces
  - Determines if move is legal
  - Makes a move (records it) and updates views
  - Supports move undo

- View shows a board and information, e.g., undo possible
  - See PuzzleView interface, implemented by application and applet

Puzzle MVC: Controller perspective

- In this example, the PuzzleController is a middleman, all communication between views and model via controller
  - 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

- In this example one controller holds all the views and executes commands in all views on behalf of model
  - Model only calls showBoard in controller
  - Some of the "intelligence" is in controller, arguably should be in model
- Controller is a candidate for refactoring

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

- ShowBoard and Undo are both *Command* classes
  - Command implemented using hook/template method

Hook method and Template Pattern

- Client code (Controller) calls a command's execute method, all Commands have such a method
  - Execute always has a view parameter
  - Execute has optional other parameter for information

- The execute method is the same in every Command, forwards to the *hook* method
  - Subclasses of ViewCommand implement hook in application specific way
    - Showing board calls appropriate method in view
    - Undo calls appropriate method in view

Lower level JButton particulars

- PuzzleGui class has panel/grid of buttons for display
  - Pressing button causes action (via controller)
  - Button is displayable, but doesn't have label
    - If button had label, it would be automatically shown
  - Instead, use setActionCommand to store command
    - Retrieved by getActionCommand, but not a label
  - Button has icon, automatically displayed

- The Icon interface (swing) implemented by ImageIcon
  - See PlainPuzzleIcon and ImagePuzzleIcon
  - An Icon doesn't typically grow, but in this application we want it to resize when app is resized

What about a “real game”

- How to make this a game? What are use cases/scenarios?

- Shuffle pieces initially, pick random piece near blank and move it, repeat 5, 10, 20 times [degree of difficulty]
  - How does this facilitate auto-solve?
  - What about showing the numbered tiles as a hint when just image puzzle used by client

- Auto-complete, we can undo string of moves, track all
  - Coalesce moves that are redundant
  - Recognize previous state of board and go back