From XP to javax.swing.*

- **What parts of embracing change can we embrace in 108?**
  - Evolutionary design, small releases, iterative enhancement
  - Simple design, don’t build for the future (will you need it?)
  - Lots of testing, testing, testing
  - Refactoring: change design, not functionality

- **What may be hard to embrace in 108?**
  - Code the test first
  - Pair Programming
  - Business aspects: meetings, customers, ...

- **Links**
  - [http://www.xprogramming.com/what_is_xp.htm](http://www.xprogramming.com/what_is_xp.htm)
  - [http://www.extremeprogramming.org/rules.html](http://www.extremeprogramming.org/rules.html)
  - [http://www.martinfowler.com/articles/designDead.html](http://www.martinfowler.com/articles/designDead.html)
Code, Design and Refactoring

(See Martin Fowler’s *Refactoring* book)

- Do some up front design: think a little before coding
  - But, refactoring can help once you’ve got working code
  - You don’t have to get the design right, you’ll change it
  - Make things simple, don’t worry too much about future

- When do you refactor? When the code smells (Fowler, Ch. 3)
  - Duplicate code: methods in one class, in two classes, similar but not identical, ...
  - Long methods (if it’s long, it’s wrong)
  - Long parameter lists
  - Speculative generality
  - Middle man
  - Inappropriate intimacy
Model, View, and Primitive Puzzle

- **Model and GUI/View communicate via inner classes/message adapters**
  - These are a kind of distributed controller in MVC
  - Often the GUI and the controller are the same thing
  - Sometimes a separate controller isn’t warranted

- **Who is responsible for button/image functionality?**
  - Consider clicking on “5”, initially in location 5, but later ...
    - In the GUI 5 is 5, but in the model it’s different
  - Who is responsible for what kind of image should be drawn in Minesweeper?
  - What happens with click/click in Bejeweled?
From Puzzle 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 in Swing (see TextFieldDemo.java)

- Nearly every Jxxyy has a model (button, list, ...)
  - Model maintains state, GUI responsible for drawing
  - GUI via java.awt.event.* also handles control, so View and Control are co-located
  - Consider JButton, what’s in state?
    - String, default, enabled, ...

- Changes in model are automatically reflected in view, not using Observer/Observable (which are clunky)
  - Consider example in ExpandableList.java, what happens when new value added to list?
  - What happens when value is selected, part of MVC?
Horstmann on JList (Core Java V. I)

The internal architecture of the list component ... is rather elegant. Unfortunately, the designers at Sun felt that they needed to show off that elegance, rather than hiding it from the programmer who just wants to use the component. You will find that the list control is somewhat awkward to use for simple cases because you need to manipulate some of the machinery that makes the general cases possible.

● How does this mesh with XP/Simplicity?
  ➢ Is there a difference between COTS and our code?
    • (Is java source commercial?)
  ➢ What are lists used for?
  ➢ Is it that hard to create a simple, expandable list?
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**
  - See **FlawedLowerTextField** and **ExpandableList**
    - In former, must use Listener, in latter can use adapter, why?

- **What about click/key modifiers, e.g., shift/control/left/both**
  - Platform independent, what about Mac?
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 aGuidisplayText(), 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
Swing Concepts

- **Object < JComponent < JContainer < JFrame**
  - The component/container pair illustrates Composite pattern: Container has Components, including other Containers
    - Understanding Z-order, painting hierarchy is important in complex applications, see Core Java/Java Tutorial/O’Reilly
  - Containers have layout managers that control how widgets are added and appear
    - BorderLayout is versatile, FlowLayout is simple, GridBagLayout is the kitchen sink, GridLayout could be useful in OOGA
- **JPanel is the simplest container, use for holding widgets**
  - Defaults to FlowLayout, add widgets and other panels, ...
Swing continued

- **JFrame is a top-level window which is a container**
  - Widgets added to a frame’s content pain, retrieved via `getContentPane()`, different from AWT
  - LayoutManager used in default `getContentPane()` is `BorderLayout`
    - Use `BorderLayout.NORTH`, not “North” (some books)
- **Layout managers illustrates the Strategy Design Pattern**
  - Encapsulates algorithm/behavior as a class, pluggable
  - Users of strategy delegate responses/use to the the strategy
  - Put a new layout in every container you use (can’t hurt?)
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