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
  - Inappropriate intimacy

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?

```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()`, 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**
  - See **FlawedLowerTextField** and **ExpandableList**
    - In former, must use Listener, in latter can use adapter, why?

- **What about click/key modifiers, e.g., shift/control/let/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 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?