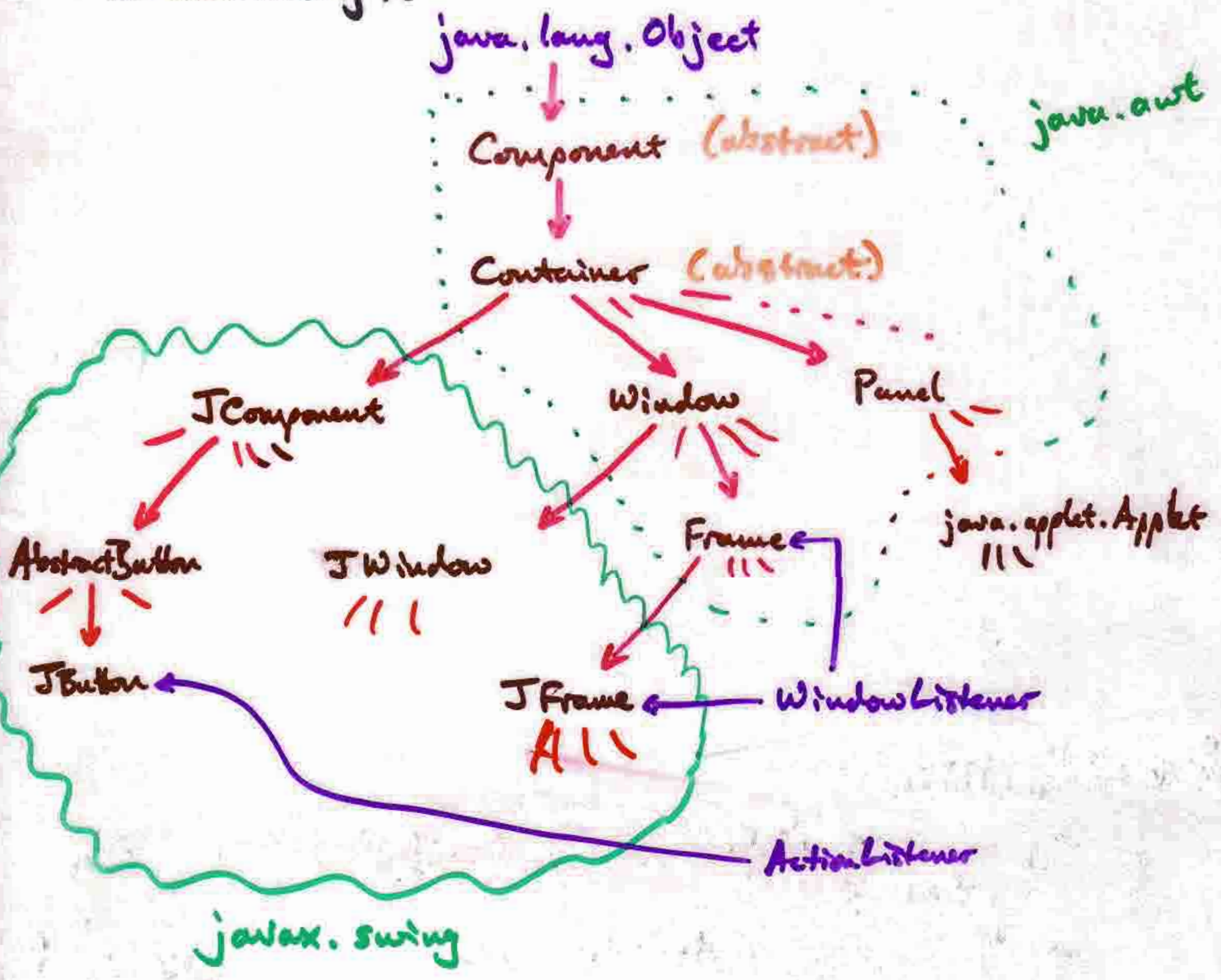
We'll say more about threads later, for now we're going to see an overview of GUIs using Swing.

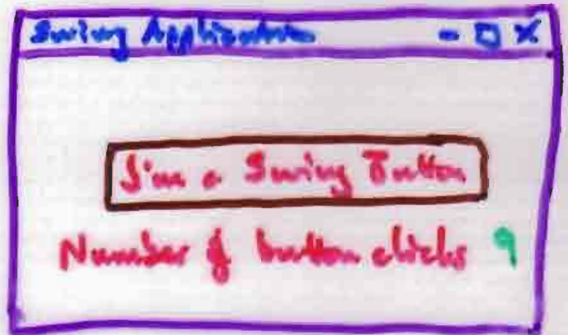
The basic steps to setting up a GUI are...

1. Import needed packages
2. Set up top level container
3. Add components and do layout
4. Set up event handling
5. Miscellaneous housekeeping!

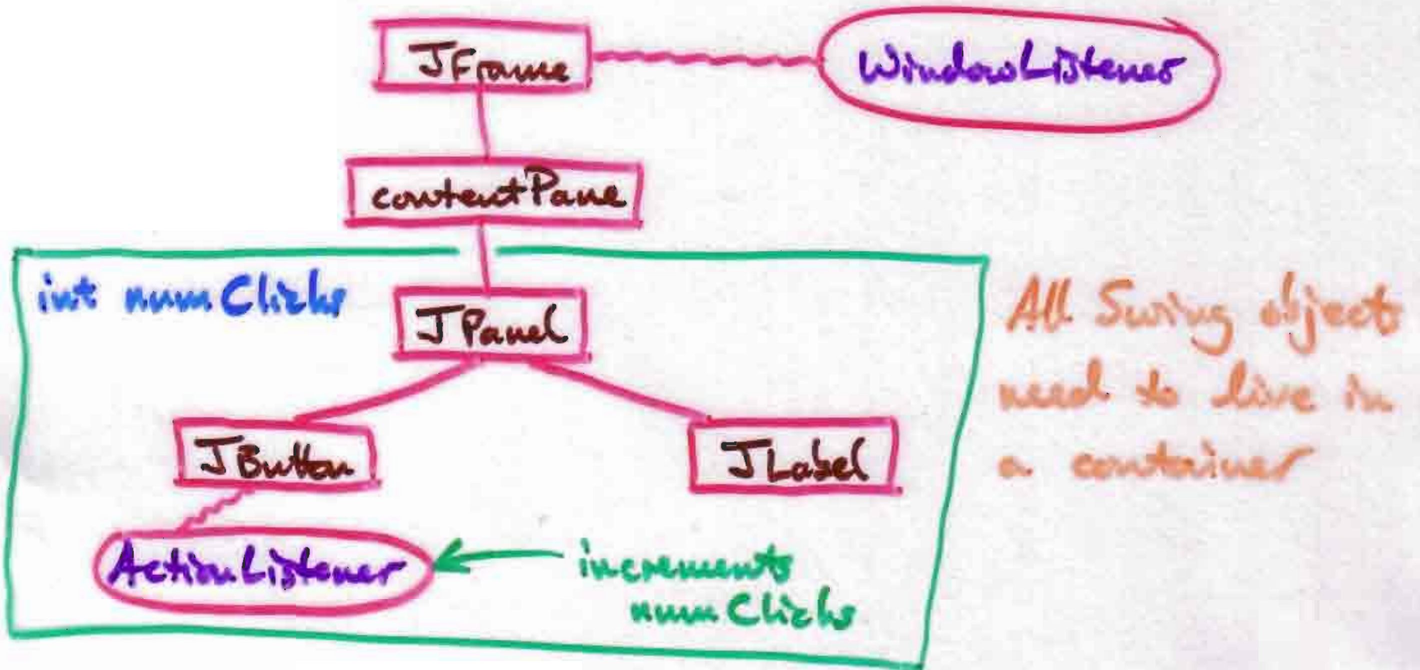
Schematically...



```java
import javax.swing.*;          //This is the final package name.
//import com.sun.java.swing.*; //Used by JDK 1.2 Beta 4 and all
                               //Swing releases before Swing 1.1 Beta 3.
import java.awt.*;
import java.awt.event.*;

public class SwingApplication {
    private static String labelPrefix = "Number of button clicks: ";
    private int numClicks = 0;

    public Component createComponents() {
        final JLabel label = new JLabel(labelPrefix + "0    ");

        JButton button = new JButton("I'm a Swing button!");
        button.setMnemonic(KeyEvent.VK_I);
        button.addActionListener(new ActionListener() {
            public void actionPerformed(ActionEvent e) {
                numClicks++;
                label.setText(labelPrefix + numClicks);
            }
        });
        label.setLabelFor(button);

        /*
         * An easy way to put space between a top-level container
         * and its contents is to put the contents in a JPanel
         * that has an "empty" border.
         */
        JPanel pane = new JPanel();
        pane.setBorder(BorderFactory.createEmptyBorder(
                                        30, //top
                                        30, //left
                                        10, //bottom
                                        30) //right
                                        );
        pane.setLayout(new GridLayout(0, 1));
        pane.add(button);
        pane.add(label);

        return pane;
    }

    public static void main(String[] args) {
        try {
            UIManager.setLookAndFeel(
                UIManager.getCrossPlatformLookAndFeelClassName());
        } catch (Exception e) { }

        //Create the top-level container and add contents to it.
        JFrame frame = new JFrame("SwingApplication");
        SwingApplication app = new SwingApplication();
        Component contents = app.createComponents();
        frame.getContentPane().add(contents, BorderLayout.CENTER);

        //Finish setting up the frame, and show it.
        frame.addWindowListener(new WindowAdapter() {
            public void windowClosing(WindowEvent e) {
                System.exit(0);
            }
        });
        frame.pack();
        frame.setVisible(true);
    }
}
```



Schematically, this could be viewed as...



Event Handling

Component or Event		Interface
JButton		ActionListener
JFrame (main Window)		WindowListener
		or WindowAdapter class
JTextField		ActionListener
JSlider		ChangeListener
Mouse Clicks		MouseListener
Mouse Moves		MouseMotionListener

Simple event handlers – implement with anonymous inner classes.

More complex handlers – implement as a separate class or as part of a larger class.

Listing this the other way round...

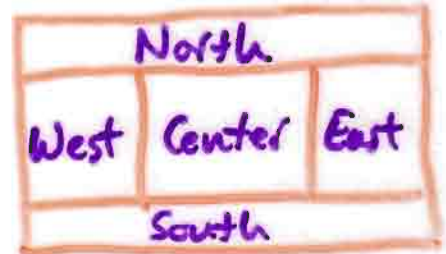
Listeners	Events generated by
ActionListener	AbstractButton JComboBox JTextField Timer
AdjustmentListener	JScrollbar
ItemListener	AbstractButton JComboBox
ComponentListener FocusListener KeyListener MouseListener MouseMotionListener	Component
ContainerListener	Container
WindowListener	Window
ChangeListener	JSlider

Layout Managers

- used instead of manually specifying component placement
- allows easy, automatic repositioning, e.g., a button or menu may change size when the text is changed.

The layout manager queries components about their size — Max, Min, and Preferred. The most common managers are...

BorderLayout — the default for contentPane



BoxLayout — single row or column, uses maximum size.

FlowLayout — the default for JPanel. Adds left to right, adding rows as needed.

GridLayout — rectangular array of equal-sized cells.

GridBagLayout — more flexible than GridLayout, though it's often easier to use nesting — put components in panels with one layout manager, then combine the panels with another.

OverlayLayout — this only centres an object in its area, typically used by JButton & its subclasses. This doesn't expand the component to fill the available space.

ScrollPaneLayout / ViewportLayout — These are used by JScrollPane and the JViewport containers.

Of course, it makes little sense talking about layout managers for objects in a container without at least a cursory description of the relationships between containers and Swing . . .

JPanel – perhaps the simplest & most versatile container, typically used to house & arrange other containers & components.

Box container – uses BoxLayout to line up components horizontally <u>or</u> vertically. Can nest.

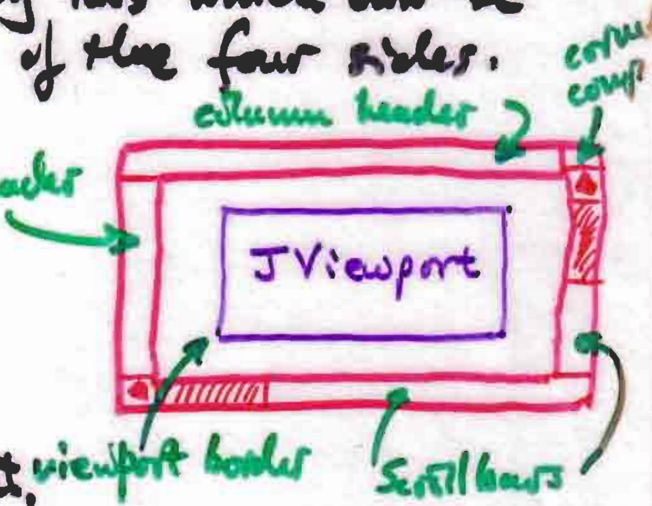
JSplitPane – holds 2 panes horizontally <u>or</u> vertically plus a repositionable divider. Uses

```
setLeftComponent()
setRightComponent()
```

or similar ones for top & bottom.

JTabbedPane – holds completely overlapping panes which are identified by tabs which can be placed on any of the four sides.

JScrollPane – holds a scrollable object for which most components can have their views set.
JViewport



JTextPane, JEditorPane, JDesktopPane are also often useful, and finally JInternalFrame is a special container of value.

It's actually useful to understand a little more deeply how Swing manages these 'panes'.

There are 5 Swing container classes which have particular importance...

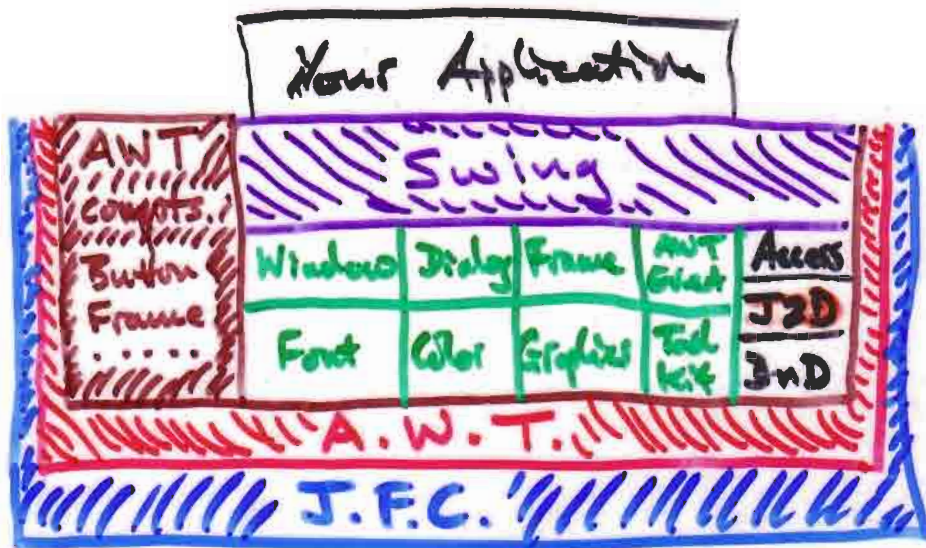
JFrame, JDialog, JWindow, JApplet, JInternalFrame

The first 4 are heavyweights, and the last is, like most Swing components, lightweight. The distinction being that the former are associated with their own native screen resources (aka peers), whereas the latter 'borrows' the screen resource of its 'ancestor'.

heavyweight	lightweight
– pixels always opaque	– pixels can be transparent
– always appear rectangular	– using transparency, can appear in varieties of shapes
– mouse events direct	– mouse events can fall through to parent
– partially platform dependent	– written in Java, so platform independent.

It's best to try to minimize mixing of these two types, especially since attempting to overlay a lightweight on top of a heavyweight will always have the heavy one on the top.

The Sun Swing developers describe the dependencies of Swing on AWT and the other Java classes using the following diagram...



java.sun.com/products/jfc/tsc/articles/getting_started

Returning to our discussion of containers, in Swing the contents of a container are stored in an intermediate structure, its content pane. So to add a component X to a container C we need to use the circumlocution...

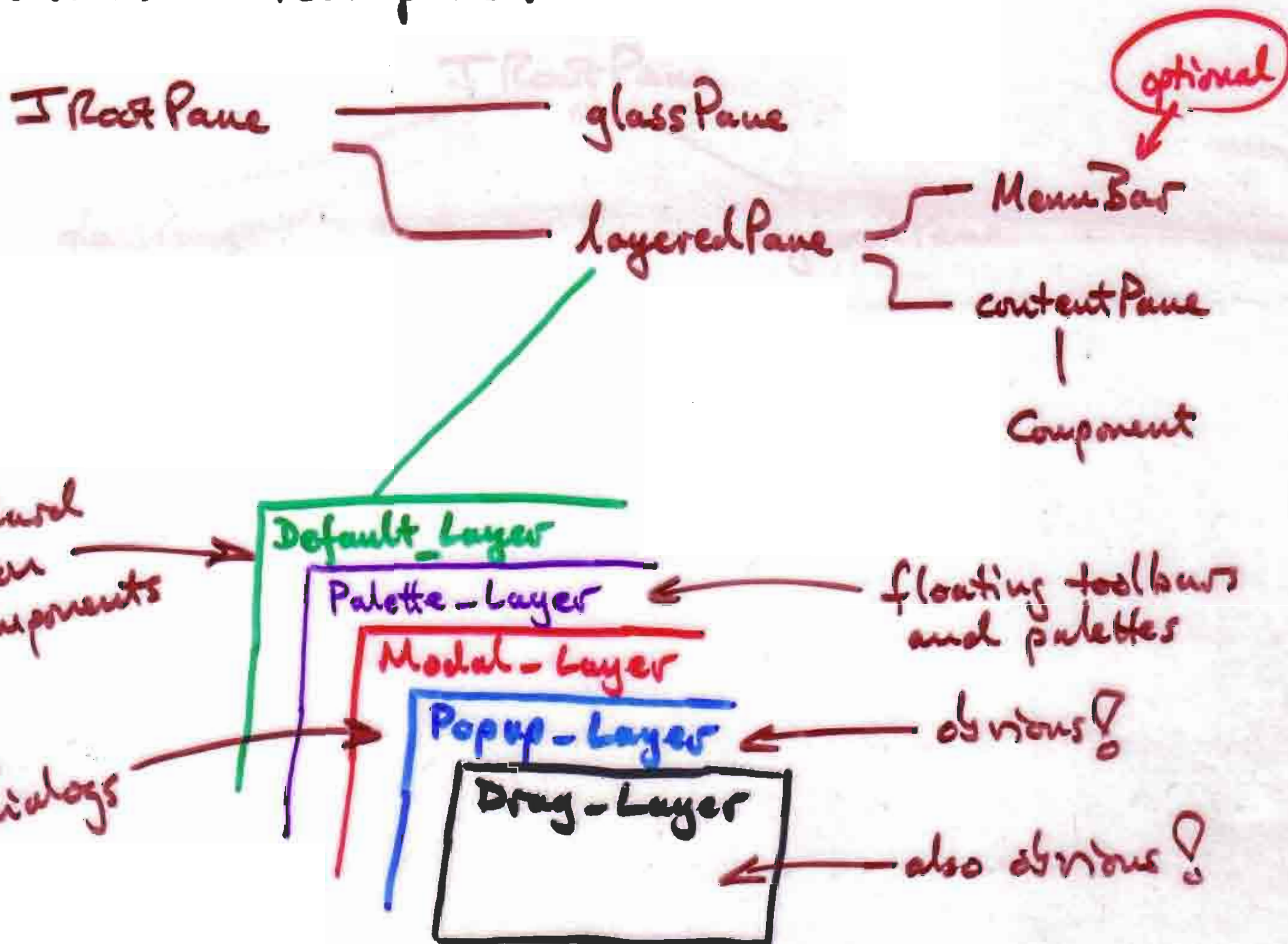
```
C.getContentPane().add(X);
```

The 5 Swing containers we mentioned before actually implement the RootPaneContainer interface, which declares methods...

```
Container  getContentPane();
Component  getGlassPane();
JLayeredPane  getLayeredPane();
JRootPane  getRootPane();
void  setContentPane(Container contentPane);
void  setGlassPane(Component glassPane);
void  setLayeredPane(JLayeredPane layeredPane);
```

so these containers actually delegate their operations to the JRootPane class.

So what's a root pane?



So containers delegate their getContentPane() to the JRootPane object they contain, which in turn delegate their version of that method to their JLayeredPane instances. There are methods (if needed) to move the relative 'heights' of these layers.

Having discussed 'containers', the natural topic would now be 'components'. Looking at the API for JComponent is itself quite helpful, but you might find it useful to start with...

java.sun.com/products/jfc/tsc/articles/component_gallery/

Broadly speaking, components fall into two groups: those which are largely 'decorative', and those which are primarily data-driven or data-gatherers. The modular ethos of Java points to a modular approach to visual applications:

- model – representing the data
- view – visual representation of the data
- controller – translating between changes in view and changes in model.

This is the underlying philosophy of JavaBeans, see

developer.java.sun.com/developer/onlineTraining/Beans/bean01.

Swing combines the view and controller (since writing generic controllers is tricky) to yield a separable model architecture, this has then 2 parts:

- model – represents the application's data.
- UI object – user-interface combination of view and controller (v-c).

For us, the v-c role is handled by the component class being used (e.g. JButton), which in turn delegates the look and feel aspects to the UI object provided by the currently installed look-and-feel. This latter can be customised to your preferences, or can mimic those of the OS.

When building GUIs, it's better that design be focussed on data rather than the UI, even though the UI is very important. The following table lists some of the component-model mappings for Swing, which has a separate model interface for each component having a logical data abstraction.

Component	Model Interface	Model Type
JButton	ButtonModel	GUI
JCheckBox	..	GUI/data
JMenu	..	GUI
JMenuItem	..	..
JComboBox	ComboBoxModel	data
JScrollBar	BoundedRangeModel	GUI/data
JSlider	..	..
JTabbedPane	SingleSelectionModel	GUI
JList	ListModel	data
..	ListSelectionModel	GUI
JTable	TableModel	data
..	TableColumnModel	GUI
JEditorPane	Document	data
JTextPane	..	..
JTextArea	..	..
JTextField	..	..
JPasswordField	..	..

Notice that this separation allows the option of plugging in your own model implementations for Swing components. Referring to the above table...

GUI-state models – define visual status of a GUI control.

application-data models – refers to data being gathered or displayed. It's very useful here to keep a clean separation between the data and the GUI.

mixed models – typically where GUI's & data need to be held continually in sync.