Research Report
Use of objects in Graphic User Interfaces in Cocoa and Java Swing Foundation classes

Wayne Buchner,6643140

13-11-2011

Contents

1 Introduction

The Graphic User Interface allows the user to interact with a program by manipulating properties of objects there by sending information from the View to the Controller. Both Objective C and Java accomplish this in similar ways. This research report focuses on the differences in producing Graphic User Interfaces using Objective C's Cocoa and Java's Swing packages. Inherently the greatest point of difference in developing on the aforementioned platform is Java Swing's ability to be a truly cross platform framework (Sparke 2006). Where as Apples Cocoa is a powerful platform centric framework constructed purely for the Mac OS and iOS platforms. This provides a focused set of frameworks.

Dias & Oliveira (2011) states that writing code can be a daunting experience for novice programmers and that designing in an interface using direct manipulation of objects is a powerful learning tool to what was once a steep learning curve. Research undertaken for this document will provide proof that advances in Graphic User Interface builders significantly reduce code writing time, increasing efficiency and reduce errors. Removing unnecessary debugging time from the development lifecycle by reducing errors and increasing code quality as GUI builders improve, may reduce development times and allow developers to concentrate on other areas. Tests results generated for this paper are the results of timed tests between Apple's native integrated development environment, XCode Oracles Netbeans. Results are the comparison in improvements to code completion and build times.

Section 2 will introduce the Graphic User Interface architecture and explain at a high level the structure of the class hierarchies including an introduction to the Model View Controller (MVC) principals. **Section ??** compares available platform generic frameworks. **Section 4** explains in detail the differences between information passing and event handling across the two platforms while.

2 Graphic User Interface Architecture An Introduction

A Graphic User Interface allows the to user interacts with a system. A well designed interface should leverage a users existing knowledge there by improving the effectiveness of the users interaction. Both Cocoa and Swing allow for the layering of objects to be built in a hierarchical manor in order for the user to perform tasks, or in programming terminology trigger events. Although they achieve this in different ways, the semantics are minute in terms of actual designing, where Cocoa generally moved towards a design process based more on WYSIWIG allowing for more productive workflow, other development languages have been slow to implement or follow this lead. Developing a Graphic User Interface in Swing can be achieved either through a plain text file containing the code for a window or via a WYSIWIG editor found in IDE's such as BlueJay, Netbeans or Eclipse.

Both platforms isolate the view from the data by implementing the Model View Controller design pattern (further discussed in Section 4) and the similarities do not end there. Both inherit all core functionalities from a root Object class, NSObject for Cocoa and the JComponent class for Swing. Briefly, although the terminology may be different, both frameworks create an interface in a similar manor. Table 1 below shows the counterpart for main GUI objects that are shared across both frameworks.

As you can see in Table 1 both platforms fundamentally build a Graphic User Interface using similar constructs. The class names may be different however the principal behind them is similar. Where Java Swing endeavors to supply a cross platform framework, Cocoa is tailored to the MacOS and iOS platforms offering a more rigid, extensible framework build foremost with the platforms specifications.

Comparative Objects	
Cocoa - NSObject	**Swing - JComponent**
Window	JRootPanel
UIView	Container, View
Subview	JPanel
UIScrollView	JScrollPane
TabBar	JTabbedPane

Table 1: Average time of 20 build cycles for a simple Hello World Program

2.1 View Hierarchy in Cocoa

Figure 1 shows Cocoa's UIKit Class Hierarchy. Note that all objects inherit directly from the root class of NSObject. Also being more platform specific it has platform centric objects targeted at specific hardware devices. Building a Graphic User Interface using Cocoa has been further simplified by the employment and development over countless iterations of the Integrated Development Environment, XCode. XCode's interface builder offers detailed information on objects and offers only objects appropriate for use in the particular project, saving the developer time implementing unsupported user interface elements. As an aside, the newest iteration of XCode allows for storyboarding of windows to further simplify navigation between windows.

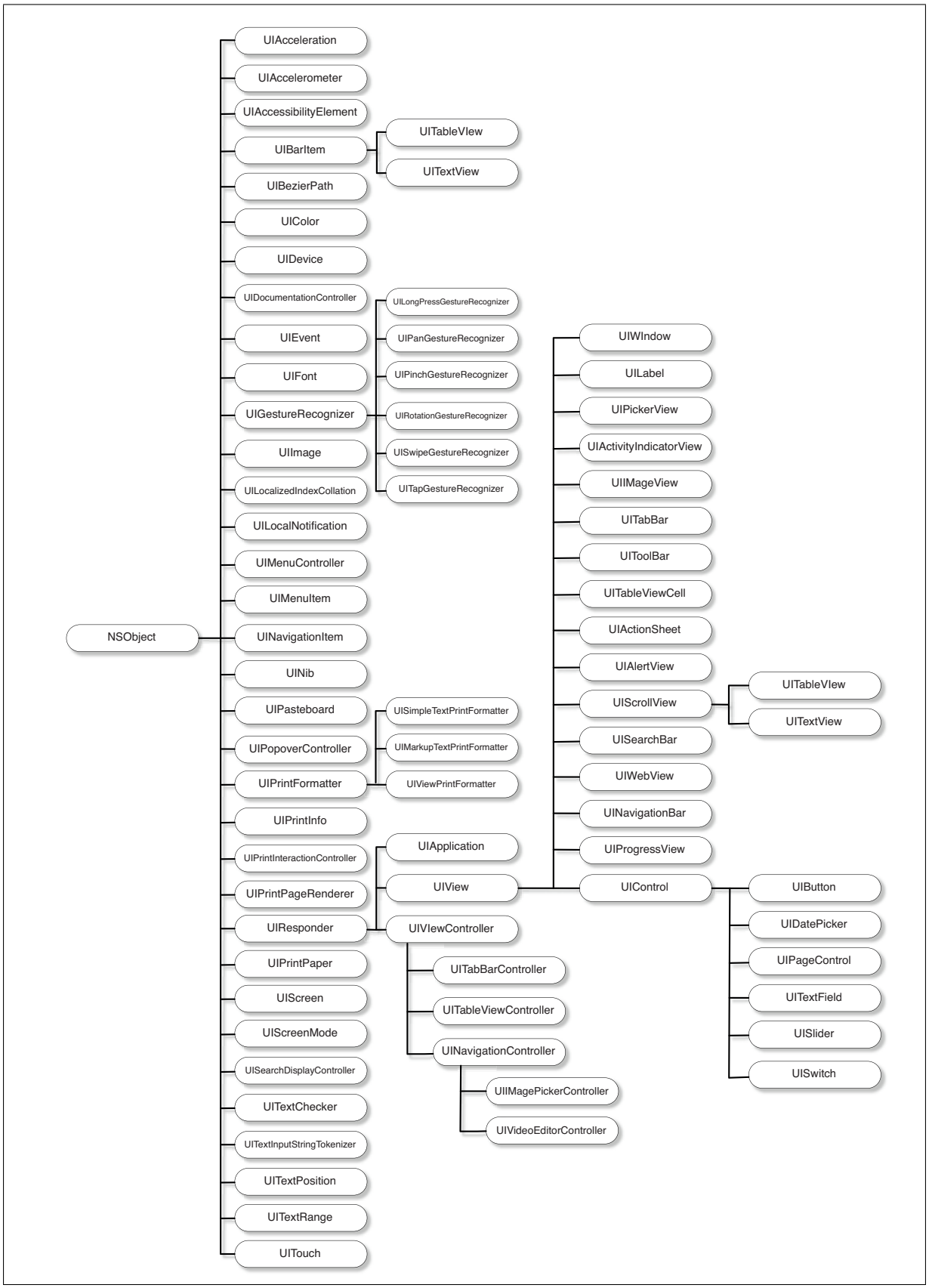

Figure 1: Cocoa UIKit Class Hierarchy. Apple (2010)

2.2 View Hierarchy in Swing

In Figure 2 you can see that Swing's hierarchy is configured only slightly differently, with JComponent inheriting from the Container class, whose root class is Object. Java Swing is a lightweight framework built on top of Java's awt framework. Due to it's completely Java construction it can be easily mixed with other Java Component class objects such as standard Java AWT labels, text fields and scrollbars.

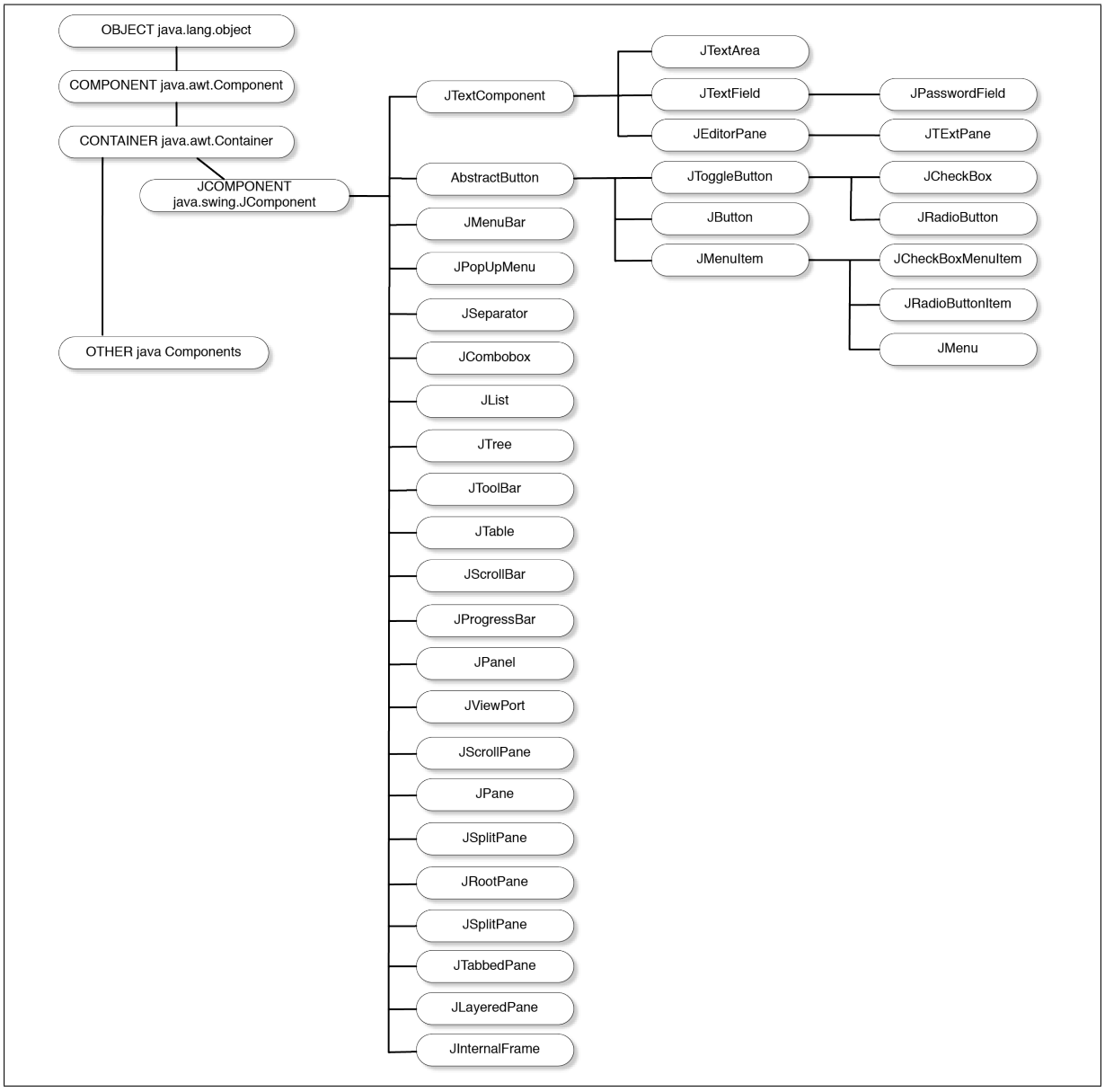

Figure 2: Swing Class Hierarchy. Oracle (2010)

2.3 Hello World

Moving from text based user interaction to a Graphic User Interface the understanding of how input is received changes. A programmer must shift from taking the result of a direct prompt to listening for user interactions.

This is called event based programming and that is how Graphic User Interface's work. Both Swing and Cocoa work in this manor. In both frameworks, **Event Listeners** respond to interaction with specific elements of the UI.

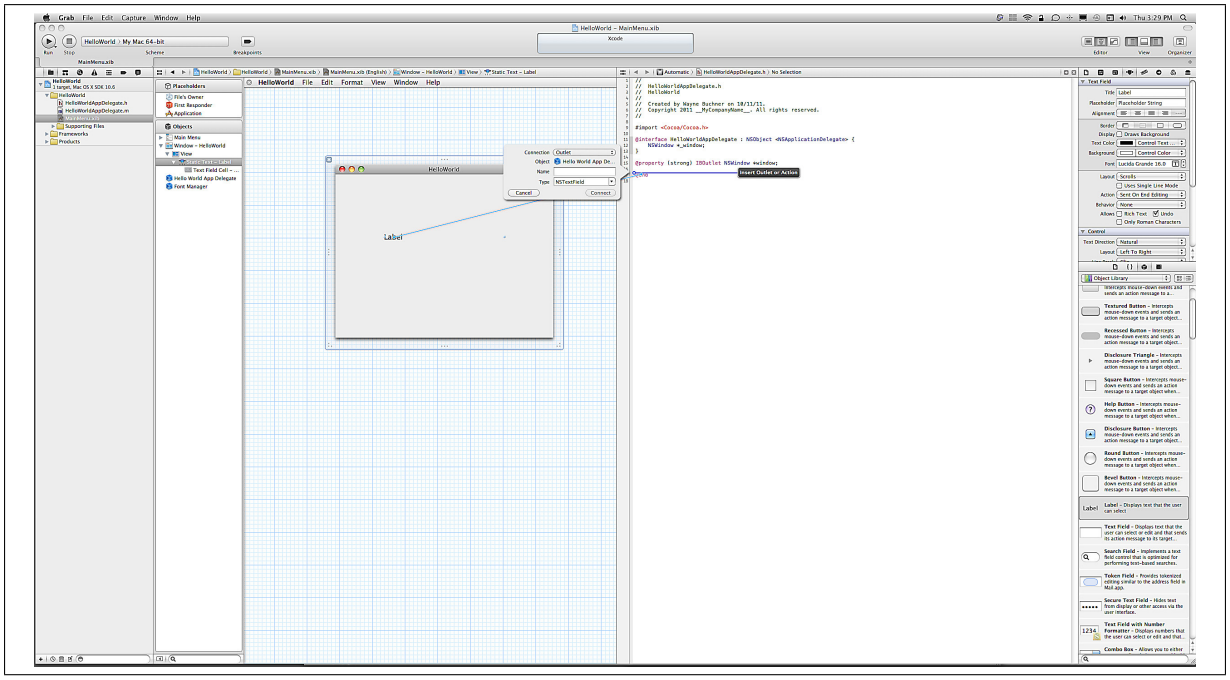

Figure 3: Interface Builder, XCode 4.1

Figure 3 shows the simplicity of **Hooking Up** an object created in interface builder to the view controller. Once an object has been created and Hooked up to a view controller or a delegate, it then becomes available to manipulate throughout the class via generated code placed directly into the corresponding header and implementation files. XCode places the correct getter and setter methods for the properties of the object via the @synthezise declaration. Java's design view works in a similar way Figure 4 shows the Design View for a Java Swing window. Selecting the object in the design view and adding an **Event** places an event listener method into the class ready for code in a similar way that XCode places the method into the implementation file.

Constructing the user interface with the assistance of a Graphic User Interface builder proved extremely fast on both platforms. Table 2 compares the user interface construction and build times for the test Hello World program using hand coding versus the IDE's assisted user interface builder. Based on the results constructing the Hello World program, developing time for a complex user interface like the interface shown in Figure 5 will be dramatically improved using an Graphic User Interface builder. The figures in Table 2 would indicate there is no noticeable difference in build time for either platform when comparing between the programs hand coded versus auto generated code. Where the figures return an indication of difference lie in the ability to efficiently design and layout a user interface.

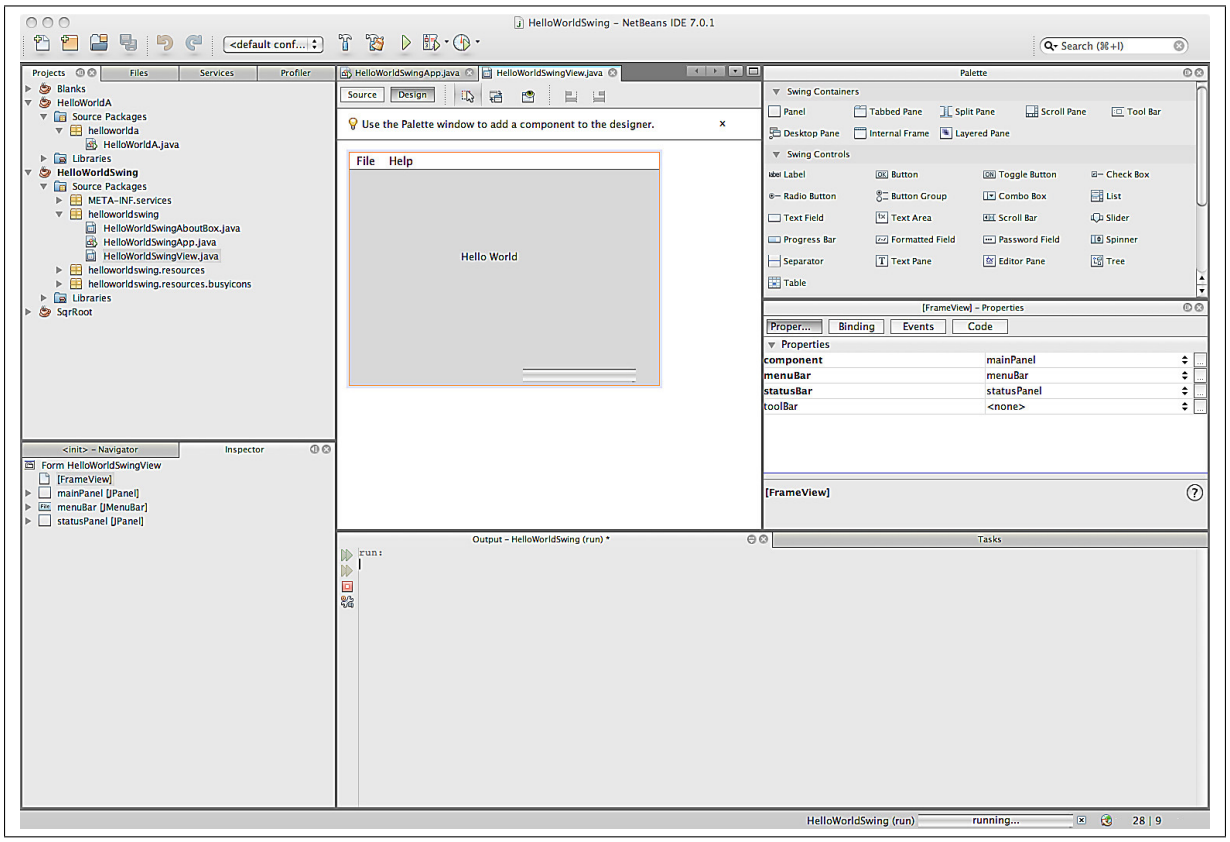

Figure 4: Design View, NetBeans 7.01

	X-Code	**X-Code** Interface builder	**X-Code** iOS	**NetBeans** (No gui builder)	**NetBeans** GUI Builder
Code Time	01:56.2	00:42.0	01:37.0	02:34.9	00:47.7
lines of code	5	1	4	10	1
Average Build Time	38.7ms	31.085ms	57.165ms		16.395ms

Table 2: Average time of 20 build cycles for a simple Hello World Program

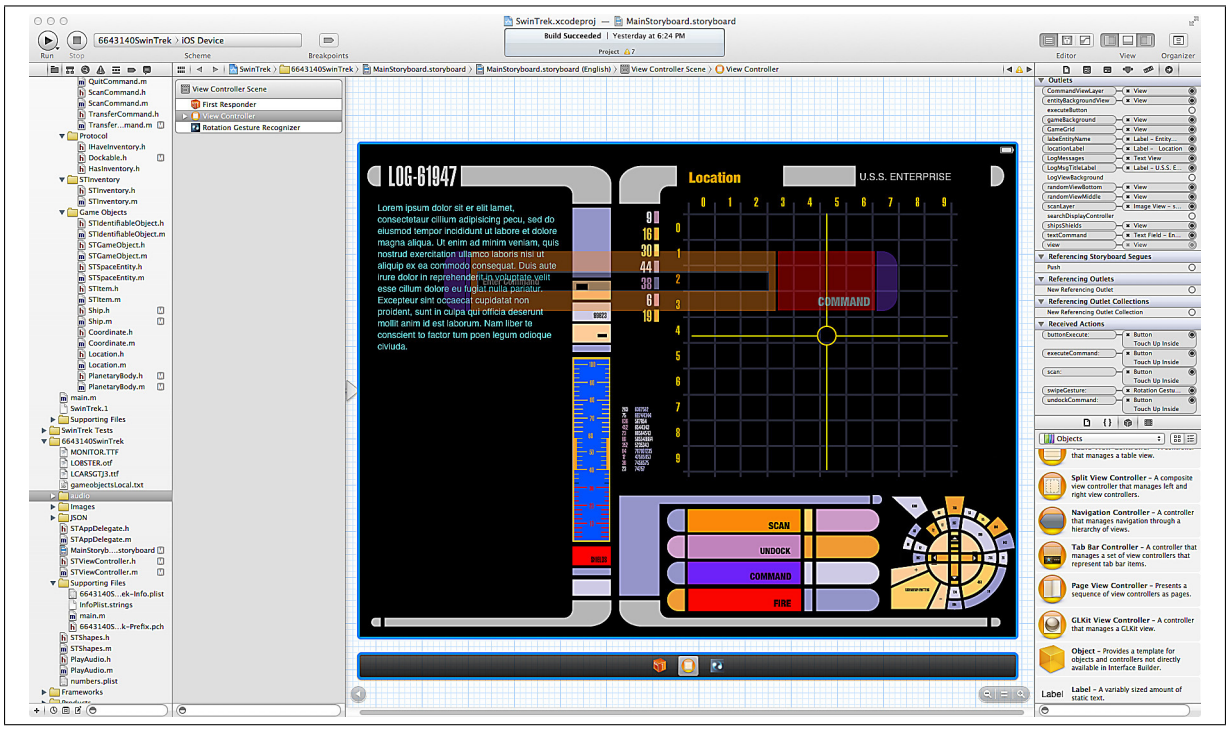

Figure 5: SwinTrek User Interface, XCode

Getting and **Setting** objects properties created using GUI builders operate similarly in both platforms. Differences across the platforms are syntactical and reserved to either taking the shortcut in XCode and allowing it to synthesize these properties values for you, or writing the getters and setter code as you would in the same way in Java. The benefit of XCode is the ability to change these post GUI build where as in NetBeans changing a property created by the GUI, the name perhaps or removing it must be done through the GUI builder.

2.4 Simple Calculator

Further testing the hypothesis that XCode is a more efficient IDE as offered in section 2.3 is proven in this section. Results of the Hello World program shown in 2.3 are further collaborated by constructing and testing a more complex program. A calculator that adds or subtracts two numbers in both Objective C and Java would be used to as further evidence. The devised test is to build four programs, a hand Coded Calculator and a Calculator created with the help of the IDE's interface builder over both platforms. This process was timed recording Build and run sequences, lines of code required and average errors per build. The results for building the simple calculators can be seen in Table 3.

Platform	Code Time	No: of Builds Required	Avg Errors/build
XCode -Interface Builder	00h:37m:00s	11	3
XCode - Hand Coded	01h:18m:00s	19	5
NetBeans - GUI Builder	01h:42m:00s	18	12
NEtBeans - Hand Coded	02h:12m:00s	37	17

Table 3: Recorded Statistics for the Simple Calculator Test

Apart from a dramatic difference in build time, laying out objects in XCode was a more efficient process. Sparke (2006) state that the most compelling reasons for using GUI builder for an application with even moderate complexity, is for efficiency. XCode's interface builder offers more affordance for the programmer and is far simpler to initialize parameters inside the IDE than NetBeans. Another point of difference where XCode allows free reign for placement of objects onto the window, Java utilizes a layout manager to specify and control the layout of objects on a JPanel or JContainer (Sparke 2006). By default Java places objects into a FlowLayout unless specified otherwise. Although object properties can be altered, Java's documentation says that it's layout manager has the final say on the size and position on components within containers (Oracle 2011). Other layout managers can be initialized if required, they are:

1. BorderLayout
2. CardLayout
3. FlowLayout
4. GridBagLayout
5. GridLayout

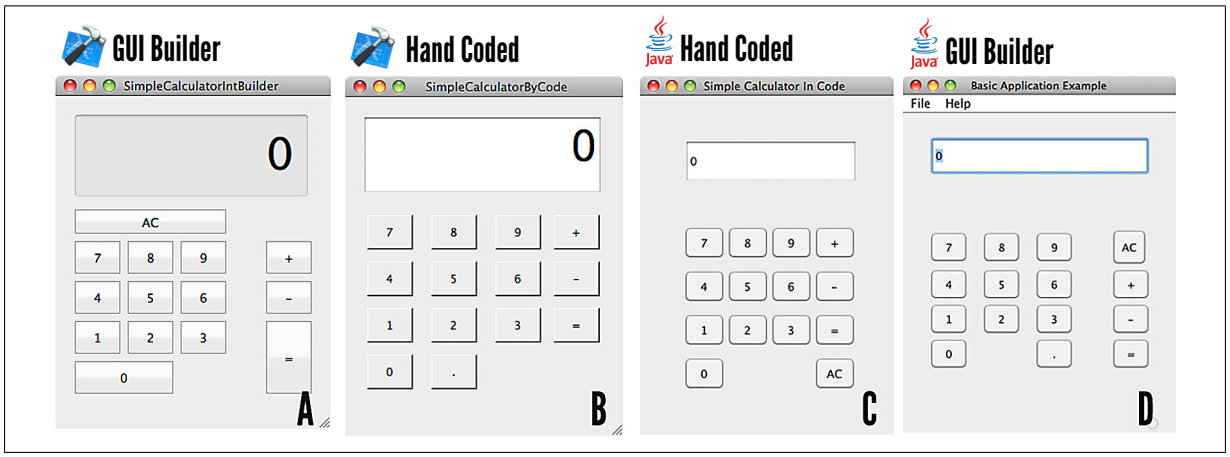

Figure 6: Four Test Calculators

Figure 6 shows the layouts of the calculators produced for the tests.

1. Calculators labeled A through B:
   - A: Built in XCode using Interface Builder
   - B: Built in XCode, hand coded.
2. Calculators labeled C through D:
   - C: Built in NetBeans using using the IDE's Interface Builder and Java Swing Packages.
   - D: Built in NetBeans, hand coded using Java Swing Packages.

Platform	Files	Lines of Code		Fields	Properties	Methods
XCode -Interface Builder	3	.h	.m	6	2	4
		19	53			
XCode - Hand Coded	3	.h	.m	6	2	6
		21	137			
NetBeans - GUI Builder	2	.App.java	.View.java	4	17	7
		44	480			
NetBeans - Hand Coded	1		.View.java	9	9	8
			163			

Table 4: Simple Calculator Comparison Results

Code practices for each platform were strictly adhered to. This allowed for the best possible comparison between the two platforms. It is clear to see from Table 4 that XCode produces the least amount of code to accomplish the result. Not only does this conform well with ISO9126(1992) standards of Good Programming Practices, future maintenance of the Objective-C's calculators will be easier than the auto generated Java code of 480 lines. Java/NetBeans creates 905% more code for it's auto generated calculator and 118% more for the hand written code. The reasons for these are outlined in Section 2.5.

2.5 Results

Objective-C allows for the multiple use of instance variable names which allows for greater code flexibility when creating dynamic object instances such as buttons. Java insists on creating each object instance with a unique name. Creating a button in Objective-C entailed one method as Listing 1. When called by the class it instantiated an instance of NSButton, added all the required properties to initialize the button and was added to the view. Further evidence of this can be seen in Figure 7 which shows class diagrams for each program. Clearly shown the Calculator created with the assistance of XCodes Interface Builder creates the most succinct class diagram. It also generates the cleanest simplest code from which to extend.

```
- (void)generateButtons:(NSView *)newView withButtonWidth:(int)buttonWidth andbuttonHeight:(int)
        buttonHeight andWhiteSpace:(int)whiteSpace withXPos:(int)xPos andYPos:(int)yPos
        withAdder:(int)adder andValue:(NSString *)stringVal
{
    NSButton *button = [[[NSButton alloc] initWithFrame:NSRectFromCGRect(CGRectMake(xPos, yPos ,
            buttonWidth, buttonHeight)) ]autorelease];
    [button setTitle:stringVal];
    [newView addSubview:button];
    [button setAction:@selector(operationPressed:)];
}
```

Listing 1: Create Button - Objective-C

Alternatively Java instead of a single method as seen above to create the required buttons for the keypad, Java required three separate methods and an array declaration as an additional class field. First a method to add a button similar to the Objective-C method of generateButton. The method shown in Listing 2 emphasizes the differences. An array of Strings was required for the instance name and syntactically the code is far more verbose. Due to the necessity of requiring additional methods to create my keypad I was required to return the button created out of this method for use by another method that added the object to the view,

titles **layOutButtons**, seen in Listing 3. This is due to the fact that Objective-C is creating a pointer to the referenced object created as the property, for example the button is allocated on the heap, so creating objects dynamically in a for loop simply points to a new memory location on the heap each time a button is created. Where any object defined outside a method in Java becomes a field of the class and declared on the stack as opposed to the heap.

```
public JButton addButton(int xPos, int yPos, int count, int i)
   {
        btnKeyPad[i] = new JButton(""+count);
        btnKeyPad[i].setSize(47, 38);
        btnKeyPad[i].setLocation(xPos,yPos);
        btnKeyPad[i].addMouseListener(new java.awt.event.MouseAdapter() {
            @Override
            public void mousePressed(java.awt.event.MouseEvent evt) {
                digitPressed(evt);
            }
        });
        return btnKeyPad[i];
   }
```

Listing 2: Create Button - Java

```
public  void layOutButtons(JFrame cn)
   {
       int xSpace = 50;
       for (int i = 1; i < 4; i ++){
           cn.add(addButton(xSpace,250,i,i));
           cn.add(addButton(xSpace,200,i+3,i+3));
           cn.add(addButton(xSpace,150,i+6,i+6));
        xSpace = xSpace+50;
       }

       cn.add(addButton(50,300,0,0));
       cn.add(addExpression(200, 150, exp[0], 0));
       cn.add(addExpression(200, 200, exp[1], 0));
       cn.add(addExpression(200, 250, exp[2], 0));
       cn.add(addExpression(200, 300, "AC", 0));
   }
```

Listing 3: Lay Out Buttons - Java

Since Objective-C combines both attributes of Static and Dynamic typing, objects can be anonymous at creation and managed at run time, as opposed to Java where the compiler is more conservative and will cause an error to ensure no run time problem arises. Cocoa's Event Handling responds differently to Java there by receiving the id of the sender from the object where the event occurred instead of implicitly sending a message saying this button was pressed. In a sense the View checks which objects was within the bounds of the event instead of Java's reacting to the touch/selection of an object on the JPanel or JContainer. Simply put, Cocoa sends the (id) of whatever object was pressed to it's Responder (event listener) from there the object can be cast into a temporary object, whereas in Java we are required to get the source of the touched object that is passed from the event handler.

Further evidence of extraneous bloated code generated by the GUI builder in Java and the extra code required for the hand coded version can be seen in Figure 7 which shows class diagrams for each program. Clearly the Calculator created with the assistance of XCodes Interface Builder creates the most succinct class diagram. It also generates the cleanest simplest code from which to extend. What makes this extra code more unique in the NetBeans IDE is that it is not editable and any changes must be made through the Design View.

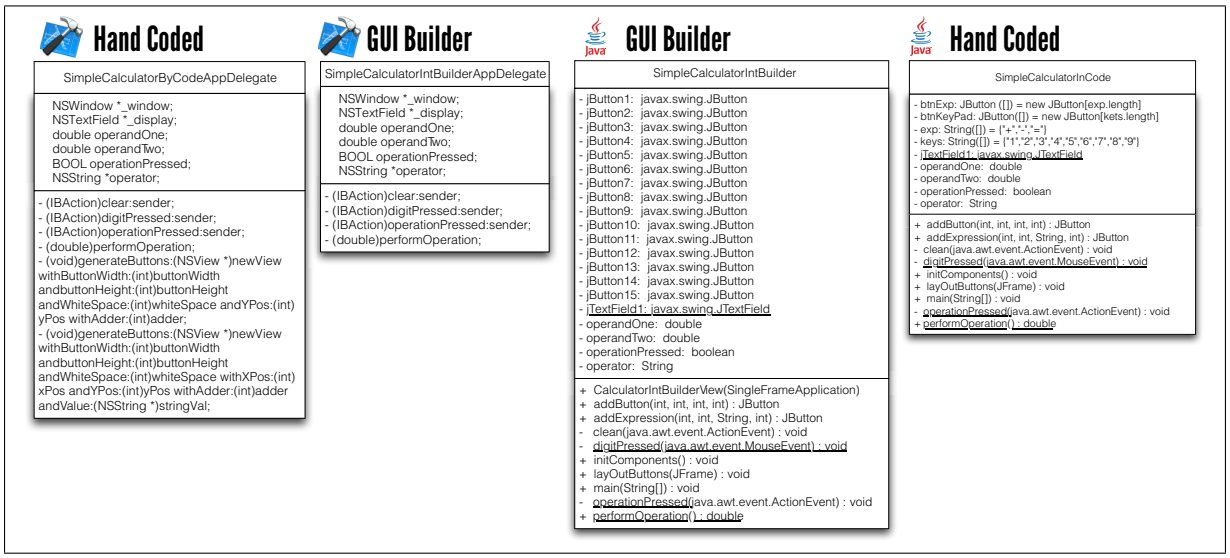

Figure 7: Class Diagram Comparison

3 Event Handling

THIS SECTION COVERS HOW THE SYSTEMS DEAL WITH EVENTS

4 Object Information Passing

References

Apple (2010), *Cocoa Fundamentals*, Apple Inc, chapter 1, p. 46.

Dias, P. & Oliveira, S. (2011), Meet and greet programming using graphical languages and tangible interfaces, *in* 'Information Systems and Technologies (CISTI), 2011 6th Iberian Conference on', pp. 1 –4.

Oracle (2010), Java 2 sdk se developer documentation, *in* 'Hierarchy For Package javax.swing'.
URL: *http://download.oracle.com/javase/1,5,0/docs/api/javax/swing/package-tree.html*

Oracle (2011), The java tutorials, *in* 'The Java Tutorials'.
URL: *http://docs.oracle.com/javase/tutorial/uiswing/layout/using.html*

Sparke, G. (2006), *The Java way : an introduction to Java programming / Gerard Sparke*, Pearson Education Australia, Frenchs Forest, N.S.W. :.