

University of
Waterloo



Faculty of Engineering

Creating Interactive User Interface Elements with Multi-layered View Hierarchy

A Report prepared for
Avoca Technologies Inc.
Richmond Hill, ON

By Si Te Feng
ID: 20474492
2B Mechatronics Engineering
September 15th, 2014

Si Te Feng
39 Mentor Blvd.
Toronto, Ontario
M2H 2M9

September 1st, 2014

William W. Melek
Director of Mechatronics Engineering
University of Waterloo
Waterloo, Ontario
N2L 3G1

Dear Professor Melek,

This work term report, entitled “Creating Interactive User Interface Elements with Multi-layered View Hierarchy”, was prepared for my 2A co-op company, Avoca Technologies Incorporated. The purpose of the report is to explain the benefits of using more view layers to create user responsive views.

For the past 4 months, I was working at Avoca Technologies as an iOS developer intern. My role was to develop new features for an iPad application called “myplayXplay”. Details of by specific tasks can be found in the introduction section of this report. Avoca Technologies provides a modern way for sport teams to stream live events onto iPad devices in real time. Live video can be both annotated and saved for playback after the event. The myplayXplay product has been used by many professional sport teams around the world for various sports such as hockey, rugby, soccer and football.

I would like to thank director of Avoca Technologies Ivan Reel, and development lead Andre I. for providing guidance and suggesting tasks for me to do throughout the term. In addition, thanks to my co-worker Ryan C. for helping to solves some of the software issues that came along. Mr. Ivan Reel can be reach at ivan.joe.reel@gmail.com for further question about the myplayXplay product. Furthermore, I would like to thank Wesley F. for proof reading this report.

This report was entirely written by me and has not received credit at any other academic institutions. I have gotten help from Apple Developer’s reference website and the “Work Term Report Guideline” document, along with other sources. These are included in the references section of this report.

Sincerely,

Si Te Feng,
2B Mechatronics Engineering

Table of Contents

List of Figures.....	iv
Summary.....	v
1. Introduction.....	1
2. Finding the Right Technique	4
3. Reducing Program Complexity	7
3.1 First Subview Layer.....	7
3.2 Multiple Subview Layers.....	9
4. Adding Code Reusability.....	12
5. Adding New Content Functionality.....	16
6. Simplified Debug Process	18
7. Conclusion	20
8. Recommendations	21
References.....	23
Appendices.....	24
Appendix A: Without Individual View Event Handling.....	24
Appendix B: With Individual View Event Handling.....	24
Appendix C: JPZoneVisualizationDataSource Declaration	25
Appendix D: Setting zoneGraphView's Data Source.....	25
Appendix E: Data Source Method Implementation	25

List of Figures

Figure 1: Live2Bench Live Video Tagging Screen.....	2
Figure 2: Simplified View Hierarchy.....	8
Figure 3: Zone Graph Screen.....	9
Figure 4: Example UITableView in iOS.....	12
Figure 5: Example for JPreorderTableView and JPTripleSwipeCell.....	13

Summary

In this report, the process of creating a layered view hierarchy is explained in terms of iOS development with Objective-C. The benefits of using more view layers is listed and compared to the drawbacks of lacking view layers.

Various challenges for designing new UI elements were assigned, such as creating a line graph, a heat graph, and a custom table view. For all three tasks, it was found that using multiple layers of views on the view hierarchy was more efficient than just using a single view layer.

It was also found that efficiency was achieved not because the initial development time was reduced, but because of the beneficial effects that a good view hierarchy provides after the initial feature development.

In general, developers should use multi-layered view hierarchy regardless of the urgency of currently development tasks. Recommendations were made for managers and developers to allocate sufficient time and attention to the design of view layers in addition to developing new features.

1. Introduction

Since the debut of the first iPhone back in 2007 [1], more and more people are using mobile devices for an increasing variety of tasks. It is not hard to see why that is the case. Smart mobile devices are lightweight, so they can be easily carried anywhere, and they accept touch inputs from the finger, which enables simple yet powerful gestures such as tapping, panning, and pinching.

In the sports industry, one can observe a similar trend for teams ranging from the ones in the NHL or NFL to local community teams. Many of the events and practice sessions are now recorded or live streamed, so that players can later review their performance by watching video playbacks. For professional teams, a statistician is often hired to analyze the rich data collected from each event and provide formal feedback to the players, so that the strength and weakness can be easily identified.

Avoca Technologies is a technology startup that allows sport teams to directly stream live video onto many iPad devices simultaneously. While the video is streaming, the live video along with many other useful information is also stored onto a local Mac Mini server located inside a suitcase to allow for video playback after an event. The core service of the company—an iPad application, is built on top of this solid hardware foundation.

Avoca Technologies needed a way to improve existing features and explore new ways that the iPad application “myplayXplay” could benefit the customers. Specifically, ways to summarize large sets of data and present them logically to the user. One can imagine that if the data can be effectively summarized without the manual work, teams could potentially cut costs, which in turn brings more business to the startup. Figure 1 shows the main tagging screen of myplayXplay iPad application.

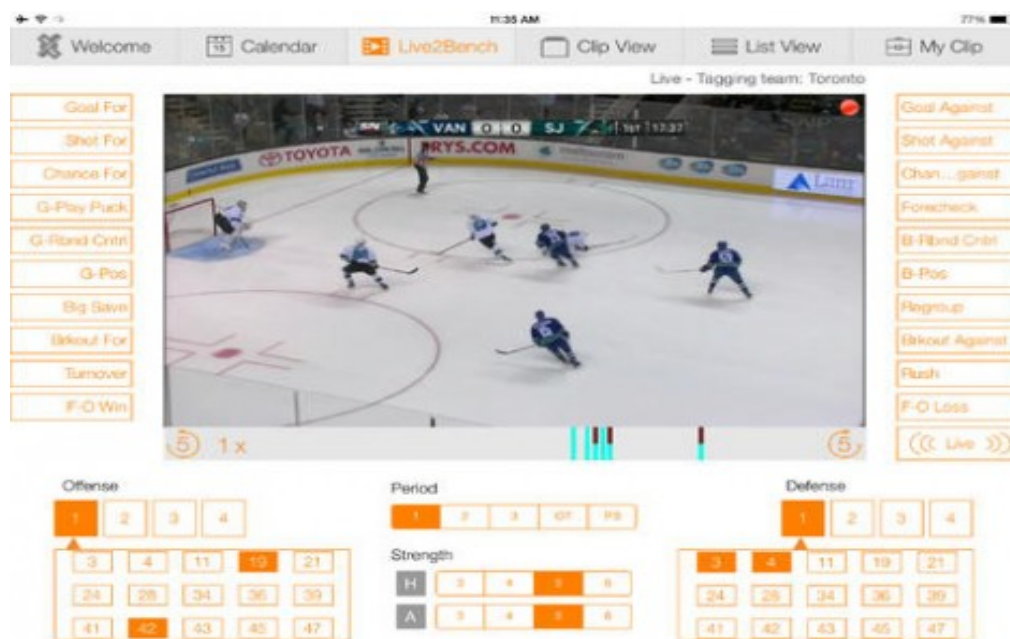


Figure 1: Live2Bench Live Video Tagging Screen

To program the iPad Application, developers at Avoca Technologies used the latest Xcode 5 IDE (Integrated Development Environment). The Xcode IDE is a native Mac OS X program that allows programmers to code on, very much like a text editor. In addition, it is also able to parse the code to check for syntax errors in the code and compile the code into an application and execute the program onto an iOS device. The language used along with Xcode was Objective-C, which is an

object-oriented language built on top of C, initially developed in the early 1980s by the Stepstone company [5].

The objective of this report is to explore many benefits of using view hierarchies when programming interactive user interface. The report will also compare these benefits to the drawbacks when view hierarchies are not layered sufficiently. Finally, it gives recommendations for future improvements.

2. Finding the Right Technique

The task of summarizing large amounts of video tag data was fairly general, so an easy solution is nowhere in sight. However, better ideas can often come up after referring to features that are already implemented. One of the existing features was tagging statistics. During a live event, the user is able to select 10 second video clips to save for later as a tag, which contains lots of relevant information such as the player number, player position, and what the player did during that time frame. On the statistics section of the iPad app, tags were grouped and displayed onto a table according to player name and the time when the tags were made. This interface allowed coach and players to easily pick the right video clip to review based on a timeline.

The player position information that goes along with each tag was particularly noticeable. If tags are made in regular intervals, then the player position was tracked throughout the game. Making a line graph that present this data visually would no doubt help in the training process. For example, if the offensive player is spending more time in the defensive zone, it gives an indication that the team might be playing too conservatively. In addition, an average indicator on the side can show the player's average field position throughout the event. After the general feasibility of the task was considered, a line graph for Soccer events was chosen to enhance the application's statistics feature.

After the line graph was considered further, it was apparent that event duration could vary greatly from 10 minutes to a few hours. To ensure graph contents could display entirely, a design constraint was put in place that required the graph to be horizontally scrollable and scalable so that the position and width of the graph could be adjusted according to user's preference. It was also found that the line graph might be cluttered if the event duration is long. A field heat graph was chosen to be the solution to the problem. The program would calculate the duration that the player spent in each of the three sections of the field (Offensive 3rd, Middle 3rd, and Defensive 3rd), and show the duration difference in different shades of color.

The final step in validating the design was to sketch a user interface (UI) design for the iPad application. This step took a short amount of time to accomplish, but was an excellent communication tool between the developers. It made sure the entire development team understands what needs to be implemented, so that no work would be lost due to misunderstanding down the road.

Building a multi-layered view hierarchy was extremely important for creating the more complicated UI elements. The line graph described above was an example of an UI element-- a piece of content that is visible to the end user. For a graph that is interactive, using multiple layers of view proved to be extremely valuable in improving the code quality. In iOS UI programming, a view can display any custom content the programmer specifies. A view could have a parent view, and

many subviews can be added into it. Each added subview is another layer of the view hierarchy.

Creating a logical view hierarchy gives lots of benefits in four major areas. It reduces programming complexities by modularizing the display contents, adds code reusability, decrease the amount of effort needed to write new features, and makes the code easier to debug in the future. When creating interactive graphs, it's essential to have the right view hierarchy to begin with so that the time of development can be greatly reduced.

3. Reducing Program Complexity

3.1 First Subview Layer

Within the Cocoa Touch UI framework for iOS, a view (UIView class) has three major roles: it is responsible to draw its own contents, handle gesture events, and act as a container for its subview instances [2]. Each view controller class contains one parent view by default. The parent view is where all the subviews are added.

If the program does not have any subview implementations, then the view controller class will need to do all the tasks that a view class would normally do. By definition, a view controller coordinates internal data and acts as the central coordinating agent for its view hierarchy [4]. It does not have any drawing capabilities other than showing and moving the views it contains. To have a view controller's root view to display all the contents required is not only unachievable in cases where custom drawing is required, but also goes directly against the concept of encapsulation. Encapsulation requires that the contents and data of one view on the screen should not have to worry about the information in other sibling views. Code that adheres to encapsulation is easy to understand and also hard to be broken.

For the task of creating a line graph, custom drawing was required. Consequently, a view class named `zoneGraphView` (subclass of `UIView`) was created and initialized in the `ZoneGraphViewController`. Subclassing `UIView` provides all the

default behavior that any view needs—adjusting the frame or hide the view—while giving programmer the ability to specify what to display inside the view. All the display contents were put inside the zoneGraphView class at this point; the view controller class itself is much cleaner than it would have been. For programmers, this initial view setup was also a good start. If the graph lines had issues render on screen, one could directly go inside the zoneGraphView drawing method to fix the issue, whereas if the graph position was not quite right, the problem could be easily fixed inside the view controller class.

Having one view that takes care of drawing the line graph was possible, but not ideal if the graph is needed to be scrollable and scalable. There were lots of benefits in using a UIScrollView subview that just takes care of the scrolling, then adding a custom graphDisplayView inside the scroll view that draws out the actual line graph. The original zoneGraphView on the other hand, drew out all the static UI elements, such as the graph axis and the position average indicator.

Figure 2 demonstrates the simplified view hierarchy.

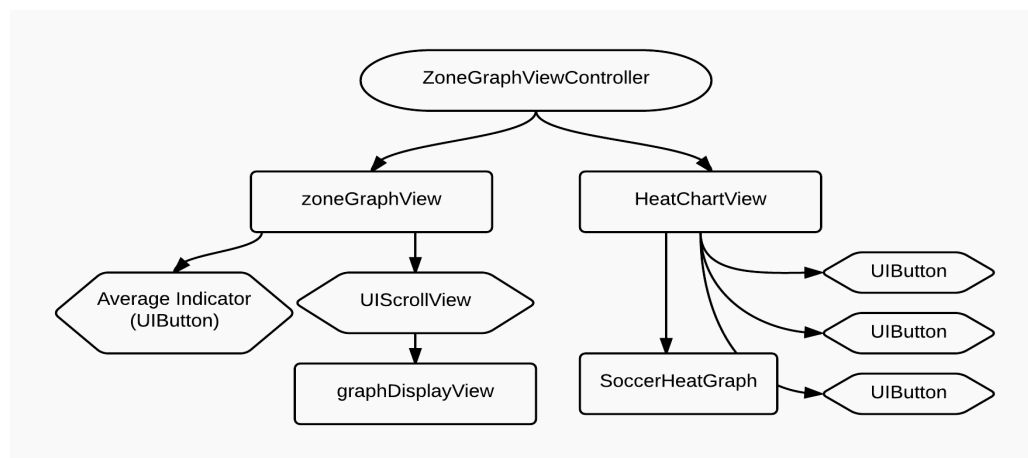


Figure 2: Simplified View Hierarchy

3.2 Multiple Subview Layers

In terms of reducing code complexity, there were three major advantages in using the extra view layers as described above.

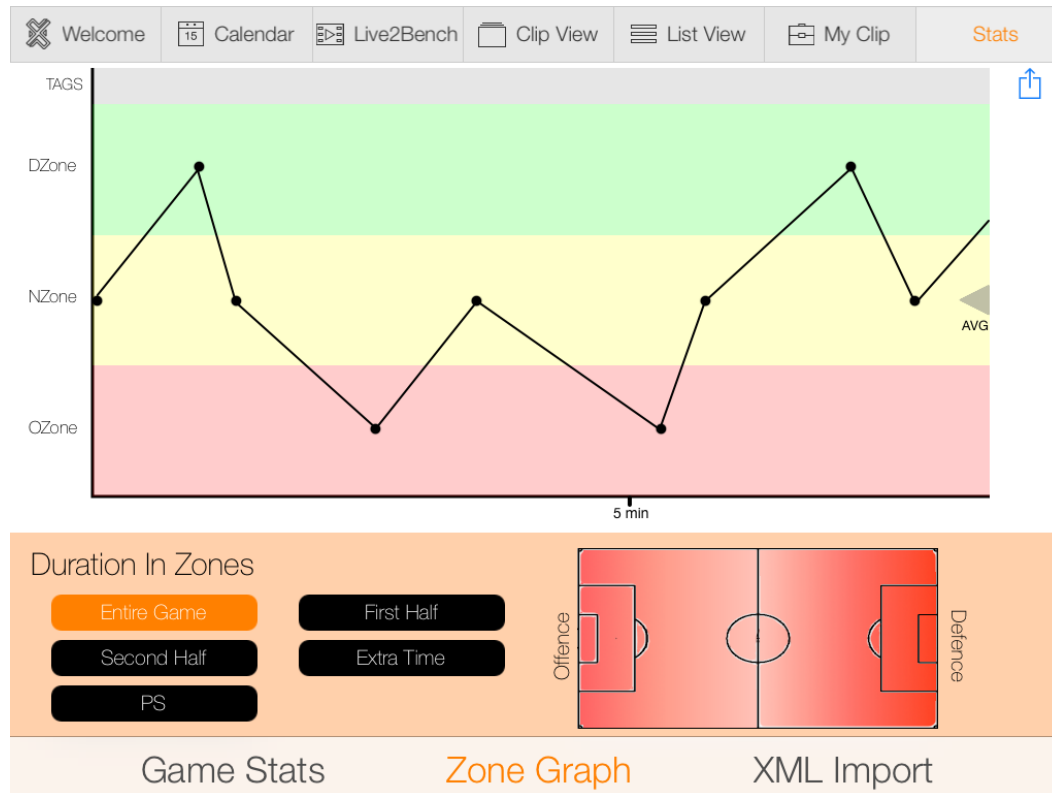


Figure 3: Zone Graph Screen

Firstly, the graph coordinate system was massively simplified for graph drawing in `graphDisplayView` (subview within the scroll view). In a typical line graph, the entire visible graph area including paddings is often wider and taller than the actual graph drawing area. In the screenshot shown in Figure 3, the graph had paddings to include axis labels. This means if `zoneGraphView` class handles graph drawing, it must account for the top and bottom offsets in every graph point calculation. This would make the code very tedious to program, not to mention

that all graph value must be inverted because the iOS coordinates system increments from top to bottom. This problem was easily fixed with the use of view hierarchy. The graphDisplayView along with its scroll view could be precisely added to the graph drawing area of the zoneGraphView class, thus eliminating the need for offsets. Inside graphDisplayView, one could then setup a macro or method that converts points from iOS coordinate system to Cartesian coordinate system. If the graph needed to shift later on, the code with layered view hierarchy would not break.

Secondly, method calls to subviews was dramatically reduced. If every single view within the line graph and heat graph were to be added directly inside the ZoneGraphViewController class, managing a group of subviews at a time cannot be achieved. As a simple example, a scenario was encountered where if the app is not playing a live event, then the screen should just be blank instead of showing the empty line graph and heat graph. One uses the “.hidden” boolean property of the UIView to show and hide the view. For example, zoneGraphView.hidden = YES. Here, one line of code effectively hides the entire line graph including the subviews within zoneGraphView. The messages propagate through the view hierarchy once a message is sent to the parent view. Without view hierarchy, methods would be sent to a great number of subviews individually, making the code repetitive and redundant.

Thirdly, event handling code structure was improved. Event handling refers to the program responding to user interactions within the app. Swiping on the screen and shaking the device are two examples of user events. When designing interactive UI, one often gives gesture callback methods meaningful names. When a user swipes on a scroll view, the corresponding method name would be “scrollViewSwipped”. If the interface is complex, then the ideal solution is for individual views to handle their own user interactions and changing display content accordingly. Please refer to the code snippets in Appendix A to see the caveats when individual view event handling does not exist.

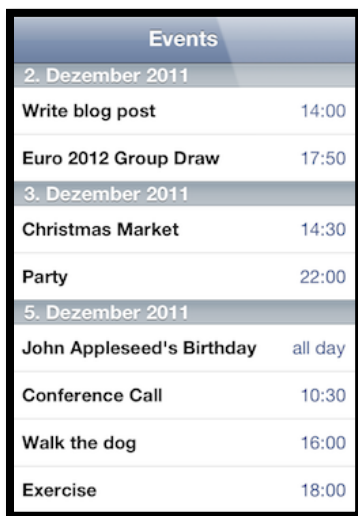
The first issue is a generic method name that gives little information for the programmer. Inside the method, the code is filled with nested if statements, which adds on another layer of obscurity. When the program finally gets to handle the user events, the method becomes extremely lengthy and hard to inspect. In addition, the program will crash if the view is a kind of class that is not expected from the programmer, and erroneously casted to a wrong class. Appendix B shows a similar situation when handled within a subview class. The result is simplified compared to the previous method.

4. Adding Code Reusability

Some UIView subclasses like the zoneGraphView were meant to be displayed in only one part of the iOS application, others however, are not. Generally, all well programed classes can be reused in different parts of the app or in different applications. Flexibility is the key to a reusable class; the class does not assume anything other than the core function it is designed to do.

Naturally, a multi-layered view hierarchy enhances the code reusability, because each subview class has its own purpose. A custom table view that was implemented on the myplayXPlay application is a great example to illustrate the concept.

The UIKit framework in iOS provides a lot of commonly used UI Element implementations, which includes the default UITableView as shown in Figure 4.



Events	
2. Dezember 2011	
Write blog post	14:00
Euro 2012 Group Draw	17:50
3. Dezember 2011	
Christmas Market	14:30
Party	22:00
5. Dezember 2011	
John Appleseed's Birthday	all day
Conference Call	10:30
Walk the dog	16:00
Exercise	18:00

Figure 4: Example UITableView in iOS

The myplayXplay iPad application has been using UITableView for a long time for selecting video clips, but its limitations were becoming an issue. The table view reordering is not configurable, and the user cannot select a row while reordering is activated. The development team decided that the user should be able to select a cell at any time, while a long press on a row would trigger reordering. A custom subclass of UITableView is the easiest way to accomplish this task, so a class named JPreorderTableView was created. Meanwhile, UI also needs to be improved through bringing new swiping ability to each table cell. Swiping left on a cell would select the row for deletion, and swiping to the right selects it for sharing. Therefore, a custom subclass of UITableViewCell was created to fit the purpose, named the JPTripleSwipeCell. Figure 5 shows an example of these two implementations.

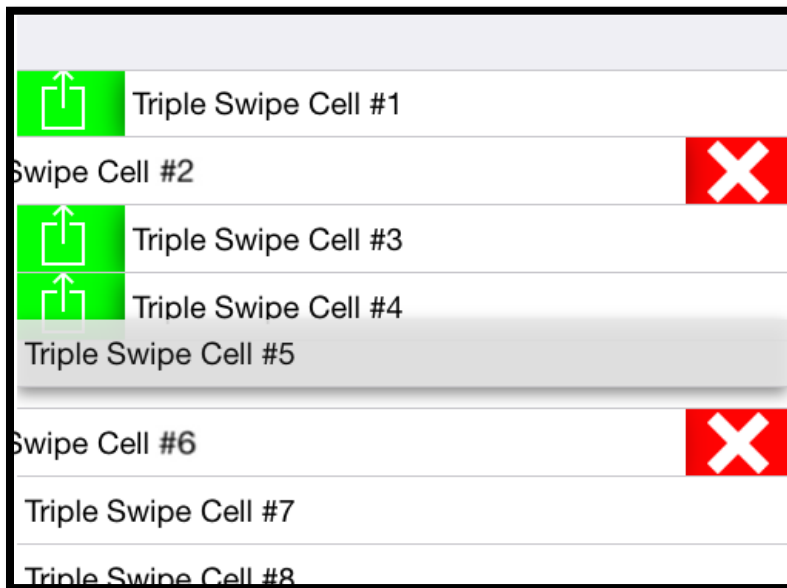


Figure 5: Example for JPreorderTableView and JPTripleSwipeCell

At first, the decision was actually to implement both reordering and swiping functions in one custom table view class. This provides one unified class to code with that can provide two separate functions, which could greatly reduce the development time. Writing less code with one less unnecessary class seems very desirable. One custom class also makes the application structure slightly simpler, with one less class to import and interface. The idea of separating these two functions came later, when more factors were considered.

It was found that eliminating JPTripleSwipeCell provides is a very little benefit when compared to the drawbacks. One hypothetical scenario that was explored was when the feature is ultimately implemented, the customers might decide they actually like the original reordering function, but want to keep the new swiping ability. In this case, the JPTripleSwipeCell class can be reused on a default UITableView, whereas a combined table and cell class must be abandoned. Linking the JPTripleSwipeCell view into JPReorderTableView hierarchy is certainly worth the extra effort in exchange for the reusability later on. On the other hand, if the new JPReorderTableView needs a regular cell instead of the swipe cells, the change is also miniscule.

Aside from making the code valuable when changes need to be made, the two custom classes provide short-term value to the application as well. Sharing and deleting saved clips screen is just one of many interfaces that the myplayXplay application contains. There are two more interfaces where a UITableView is used.

However, these two interfaces had their own custom cells already and did not require sharing ability. It seemed very beneficial for the application to implement the new reordering function on these two interfaces as well in the near future. The separated JPReorderTableView class that has been implemented could directly substitute the default UITableView, where the new feature would be instantly applied.

The resulting application with JPReorderTableView implementation was not only easily adaptable for future changes, but also flexible enough to be used in other parts of the myplayXplay application. This would save lots of development time and cost in the future.

5. Adding New Content Functionality

When the heat graph (SoccerHeatGraph) was first being implemented, it contained nothing more than a soccer field image as a placeholder. The initial challenge was to decide whether to put an empty container than encloses the heat graph, or add the graph directly to the root view of the view controller. An empty container view was ultimately added for expandability. A HeatChartView subview class was initialized and added to the view controller first, and then the SoccerHeatGraph was added to the HeatChartView, as shown in Figure 3.

Soon after, the layered structure was proven to be worthwhile. A new enhancement was needed to allow user to specify which half of the soccer game he/she wants to see on the heat graph. A few buttons was added to the HeatChartView while SoccerHeatGraph was shifted to the side. The amazing part of this setup was the new button controls could directly send callback methods within HeatChartView, telling it which half of the game user selected, and the HeatChartView in turn relays that information to SoccerHeatGraph to change the gradient display. This relay control was simple and seamless, yet keeping SoccerHeatGraph modular at the same time for reusability.

For the view controller that contains the HeatChartView, there were absolutely no changes, another convenience when integrating new features. The buttons were added and linked within the HeatChartView class itself, without the need to go

into any other classes. This makes sense because the enhancement is made for HeatChartView after all.

The ability to easily added new features to existing class is a benefit that layered view hierarchy can easily provide. In general, if the UI elements being designed have the potential to acquire new features in the future, the more layered view hierarchy should be used over a simpler view hierarchy.

6. Simplified Debug Process

Creating subviews also helps with the debugging process when bugs are found in the software. Besides managing views, a view controller is the source of information for all of its subviews. With the help of subviews, a view controller only needs to focus on providing data instead of also manipulating its data for display purposes. This makes the view controller class well organized so that a data error can be found with ease even when the data is complex and dynamic, given that the view controller is implementing a proper protocol for its subviews when necessary.

Information contained within the view controller is either directly passed onto the subviews at the time of instantiation, right after instantiation, or through the use of a protocol in Objective-C.

Protocols are generally used when the amount of information that the subview requires is large. In the case of `zoneGraphView`, it shows a large number of position data for a live event, so it is an easy decision to create a `JPZoneVisualizationDataSource` protocol. Please refer to Append C for the protocol method declarations.

The methods defined in this protocol can be seen as the questions that `zoneGraphView` wish to ask. The “@required” and “@optional” compiler flags indicate whether the questions need to be answered. When

ZoneGraphViewController add zoneGraphView to its main view, it must also set the data source property of zoneGraphView. Please refer to the example shown in Appendix D.

Just after the first iteration of zoneGraphView was created, there were lots of compile time issues such as graph lines being skewed or not showing up at all. The problem was approached through validating the initial data. Since the zoneGraphView protocol methods was already in place, data could be easily verified in “numberOfDataPointsInGraphView” method implementation within ZoneGraphViewController class. Please refer to the method implementation in Appendix E.

The issue was immediately identified as missing info inside “self.allTagInfo” array. This is one of many bugs that were encountered during the graph implementation, but it illustrates the simplicity of debugging with protocols.

7. Conclusion

Among many of its benefits, a multi-layered view hierarchy reduces complexities in each class, adds code reusability, simplifies new feature integration, and makes the code easier to debug. Thus, adding more layered subviews is beneficial for an application.

Furthermore, user interface elements should be designed with both flexibility and expandability in mind. When adding custom views onto the view hierarchy, one should ensure that the amount of layers is enough to meet current and future demands. A good view hierarchy implementation is critical to an application's development process because choosing the wrong view structure hinders application development progress down the road.

8. Recommendations

Creating a reasonable view hierarchy greatly affects the application in general, but is often neglected by management. Developers should raise the awareness that view layer structure design is just as important as programming new features, if not more so. When a project deadline is planned, both development and management teams should account for the time required to construct a well-designed view hierarchy in addition to the new feature implementation. The following recommendations are made based on the idea.

During weekly or daily development meetings, programmers should present their UI development plans to others if relevant. The team would then have a short discussion on how well the view layers are constructed to ensure development efficiency. If anyone finds any flaws in the plan, the plan can be improved immediately without any cost to the company.

At times, features need to be developed as soon as possible due to urgent requests from the customers. Developers need to understand that this should not compromise the quality of the code structure itself. One should still follow the standard way for developing a layered view hierarchy instead of putting all subviews into one class for simplicity. Some difficult bugs might exist if time is urgent. However, fixing bugs on a well-constructed code base is much better than fixing the entire view hierarchy when changes are required.

For each view class that implements a particular part of the UI, the programmer should use code comments to explain the use of that class and where the view can be found when using the application. In addition, view classes should be put into meaningful folders within the “Project Navigator” to stay organized. This way, when other developers need to reuse a particular class, she/he can find the right class to use and learn its exact use case right away.

References

1. "Apple Reinvents the Phone with iPhone.", [website], Apple Press Info. Apple Inc., 9 Jan. 2007. Web. 1 Sept. 2014.
<<https://www.apple.com/pr/library/2007/01/09Apple-Reinvents-the-Phone-with-iPhone.html>>.
2. "Working with the View Hierarchy." , [website], Mac Developer Library. Apple Inc., 8 Aug. 2013. Web. 1 Sept. 2014.
<<https://developer.apple.com/library/mac/documentation/Cocoa/Conceptual/CocoaViewsGuide/WorkingWithAViewHierarchy/WorkingWithAViewHierarchy.html>>.
3. Cedrone, Kevin. "Work Report Writing Guidelines." , [website], [https://uwaterloo.ca/mechanical-mechatronics-engineering/sites/ca.mechanical-mechatronics-engineering/files/uploads/files/Work Report Guidelines for Website.pdf](https://uwaterloo.ca/mechanical-mechatronics-engineering/sites/ca.mechanical-mechatronics-engineering/files/uploads/files/Work%20Report%20Guidelines%20for%20Website.pdf). University of Waterloo, 8 Aug. 2012. Web. 28 Aug. 2014.
<[https://uwaterloo.ca/mechanical-mechatronics-engineering/sites/ca.mechanical-mechatronics-engineering/files/uploads/files/Work Report Guidelines for website.pdf](https://uwaterloo.ca/mechanical-mechatronics-engineering/sites/ca.mechanical-mechatronics-engineering/files/uploads/files/Work%20Report%20Guidelines%20for%20Website.pdf)>.
4. "About View Controllers." , [website], IOS Developer Library. Apple Inc., 13 Dec. 2012. Web. 4 Sept. 2014.
<<https://developer.apple.com/library/ios/featuredarticles/viewcontrollerpgf-oriphoneos/Introduction/Introduction.html>>.
5. "Why Objective-C?" , [website], IOS Developer Library. Apple Inc., 15 Nov. 2010. Web. 2 Sept. 2014.
<https://developer.apple.com/library/ios/documentation/Cocoa/Conceptual/OOP_ObjC/Articles/ooWhy.html>.

Appendices

Appendix A: Without Individual View Event Handling

```
- (void)someViewSwipped: (UITapGestureRecognizer*)recognizer
{
    UIView* generalView = recognizer.view;

    if([generalView isKindOfClass:[graphDisplayView class]])
    {
        graphDisplayView* displayView = (graphDisplayView*)generalView;
        if(rec.state == UIGestureRecognizerStateBegan)
        {
            [displayView respondToTapMethod1];
            [displayView respondToTapMethod2];
            .....
        }
    }
    else if([generalView isKindOfClass:[zoneGraphView class]])
    {
        zoneGraphView* zoneGraph = (zoneGraphView*)generalView;
        if(rec.state == UIGestureRecognizerStateBegan)
        {
            [zoneGraph respondToTapMethod1];
            [zoneGraph respondToTapMethod2];
            .....
        }
    }
    ..... (any other subviews that responds to tap)

    else
    {
        NSLog(@"error");
    }
}
```

Appendix B: With Individual View Event Handling

```
- (void)zoneGraphTapped: (UITapGestureRecognizer*)recognizer
{
    zoneGraphView* zoneGraph = recognizer.view;
    if(recognizer.state == UIGestureRecognizerStateBegan)
    {
        [zoneGraph respondToTapMethod1];
        [zoneGraph respondToTapMethod2];
        .....
    }
}
```

Appendix C: JPZoneVisualizationDataSource Declaration

```
@protocol JPZoneVisualizationDataSource <NSObject>

@required
- (NSUInteger)numberOfDataPointsInGraphView: (graphDisplayView*)graph;
- (JPZonePoint)graphView: (graphDisplayView*)graph
zonePointForPointNumber: (NSUInteger)number;
- (NSUInteger)numberOfTagsInGraphView: (graphDisplayView*)graph;
- (NSDictionary*)graphView: (graphDisplayView*)graph
tagInfoDictForTagNumber: (NSUInteger)tagNum;
- (CGFloat)eventDuration;

@optional
- (CGFloat)timeForPeriodEnded: (NSUInteger)period;
- (NSInteger)numberOfPeriods;
- (NSInteger)numberOfTaggedPeriods;
- (NSString*)nameForAllTaggedPeriod: (NSUInteger)period;

@end
```

Appendix D: Setting zoneGraphView's Data Source

```
//Inside zoneGraphView.h
@interface zoneGraphView : UIView
@property (nonatomic, strong) id<JPZoneVisualizationDataSource>
dataSource;
@end

//Inside ZoneGraphViewController.m
- (void)viewDidLoad
{
    [super viewDidLoad];
    .....
    self.graphView = [[zoneGraphView alloc] initWithFrame:CGRectMake(0,
25, 1024, 500)];
    self.graphView.dataSource = self;
    [self.view addSubview:self.graphView];
    .....
}
```

Appendix E: Data Source Method Implementation

```
//Data source implementation
- (NSUInteger)numberOfTagsInGraphView: (graphDisplayView*)graph
{
    NSUInteger numberOfTags = [self.allTagInfo count];
    return numberOfTags;
}
```