University of Waterloo

Faculty of Engineering

Building Mobile Applications with Functional Reactive Programming and MVVM Design Pattern

A report prepared for

pebble

Pebble Technology Corp.
Palo Alto, California

By
Si Te Feng
3A Mechatronics Engineering

May 4th, 2015

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

May 4th, 2015

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

Dear Professor Melek,

This work term report “Building Mobile Applications with Functional Reactive Programming and MVVM design pattern” was prepared for my 2B co-op company, Pebble Technology Corporation. The purpose of the report is to show how I used MVVM design pattern and ReactiveCocoa framework to make the Pebble iOS application more concise and structured.

Previous coop term, I was an iOS developer at Pebble mobile team. My role was to develop new features for the new Pebble iOS 3.0 iPhone app that’s launch on May 15th. Pebble Technology is a smartwatch design company that focuses on developing e-paper display watches. The mobile apps on iOS and Android play a pivotal role in supplying information to the Pebble watch through Bluetooth. Lots of people use Pebble to quickly glance at notifications or control the music on their phones.

I would like to thank iOS team lead Daniel for providing valuable help throughout the term along with many late nights reviewing my pull requests. Also I want to thank everyone for making Pebble better every day.

This report was entirely written by me and has not received credit at any other academic institutions. I have gotten help from the “Work Term Report Guideline” document along with other sources. All sources are listed in the Reference section of this report.

Sincerely,

Si Te Feng,
3A Mechatronics Engineering

Table of Contents

List of Figures

Summary

In this report, the age-old problem of massive code size for a given view controller class was explained, followed by a detailed analysis on a new software architectural design pattern called MVVM that could solve the problem.

One major pre-requisite for MVVM, Functional Reactive Programming, was then introduced. The details on how the design pattern can be applied in practice were shown. The implementation details including the ReactiveCocoa framework for iOS along with the reasons for choosing the framework were also described in detail. It was noted that ReactiveCocoa has its own advantages and drawbacks, but was ultimately chosen for its conciseness and ease of development.

In order to measure the effectiveness of ReactiveCocoa, six categories in programming were explored, which includes code clarity, ease of development, performance, memory management, debugging, and testing. The use of ReactiveCocoa in each category was analyzed in this report.

It was concluded that by incorporating MVVM and ReactiveCocoa in large mobile projects such as the Pebble iOS app, long-term efficiency in software development could be acquired. Finally, recommendations were made for

using new languages and tools to make the process even more efficient in the future.

1.0 Introduction

It all started 7 years ago when a University of Waterloo engineering student Eric Migicovsky founded a company called Allerta that makes smartwatches that pair with Blackberry phones. After 3 years with modest progress selling the InPulse smartwatches, Allerta started to lose its market traction due to the increased popularity of iOS and Android mobile platforms. Eric along with his 4 classmates then decided to design a brand new watch from scratch that would connect with iOS and Android devices to serve the bigger market. Pebble Technology was born.

Needless to say that Pebble was a whopping success when it was announced on Kickstarter back in 2012. Within 37 days, the project raised $10,266,845, becoming the most funded project ever on the platform. [1]

This year, Pebble once again surprised Kickstarter with the Pebble Time smartwatch. The new version has a colored e-paper display, a built in microphone for voice replies, a charging port that supports smartstraps, has a 20% thinner body than the previous generation, while maintaining the same 7 days of battery life. This time, Pebble has raised $20,338,986, twice the amount as the previous campaign and reclaimed the number one spot on Kickstarter. [2] Figure 1 shows what the new Pebble Time looks like.

Figure 1: Pebble Time banner on Kickstarter

While the hardware improvements are appealing, the majority of Pebble's efforts have been put into delivering a smooth software experience. For version 3.0, the Pebble OS firmware team worked tirelessly to incorporate colorful animations, while the mobile team implemented many new features along with a fresh new user interface.

In the beginning of the term, the total number of iOS developers at Pebble was one. In fact, the entire mobile team comprised of three full time developers. When the two interns came in on January 3rd, the size of the mobile team basically doubled overnight. The iOS team works independently from its Android counterpart, the similarity of the two mobile apps ends on the user interface level. Both teams communicate with the firmware team to come up with a reasonable software specification in the beginning and then implement the functions based on those decisions. The design team was also a great help in creating the image assets needed for the revamped UI. The acronym "UI" is a common short form for "user interface".

Each iOS developer at Pebble uses the Xcode 6 IDE (Integrated Development Environment) on a Macbook Pro 15'' laptop computer. The Instruments application was used infrequently to detect memory allocations and leaks. The language used for programming is Objective-C.

The task for the iOS team was to finish building the new iPhone app by May 15th, 2015, so that the Kickstarter backers can start using the companion app by the time they receive their Pebble Time watches. The challenge was to write maintainable and clear code within a short time frame, so that the app can be delivered on time. The problem that hinders the progress lies within the traditional iOS MVC design pattern.

MVC stands for Model-View-Controller. It's a software architectural pattern that helps to organize classes into three main categories within the realm of object-oriented programming. Almost all UI developers start by implementing the MVC pattern due to its simplicity and popularity. It was created in 1978 by Trygve Reenskaug for the Smalltalk language, and was used in many other languages ever since.[3] It was widely adopted in the Cocoa Touch UI framework for iOS. This pattern is popular for its code reusability as well as simplicity. The controller consists of one class, which is the subclass of the abstract UIViewController. The single controller class controls both the views that the users see as well as the underlying models that represent the data. For simple iOS applications, this pattern works exceptionally well, however, for production apps such as Pebble iOS, the controller class often gets bloated into more than 500 lines of code and becomes unmanageable. The controller code is also very hard to unit test because all the UI related code is intertwined with the logic code.

This report was written to reveal the process of finding the right solution to the problem stated above as well as explain why the combination of Functional Reactive Programming and MVVM design pattern is the best solution.

2.0 Reducing code in controller classes

A common method that programmers use to split a complex class into smaller pieces is through inheritance. This concept allows the child class to inherit all the features from its parent, thus creating a class hierarchy.

iOS programmers who created an universal app before are familiar with this pattern. Universal app refers to an app that either runs on the iPhone or iPad after being compiled for the platform. One can simply create an abstract super UIViewController class that implements logic methods, and create two concrete controller classes that inherit from the super class, one for iPhone, and one for iPad. Unfortunately, it was quickly found that this approach leaves lots of room for loopholes.

The split between the parent and child controller classes are often very blurry. The abstract super class in theory should not have any UI specific method invocations, while the child class should only contain UI code to allow for simplified unit testing. Nonetheless, there is nothing that can prevent the programmer to accidentally include a line of UI code inside the abstract super class. For example, if an app for both platforms contains auto-scrolling banners, one might be tempted to start the scrolling banner in the abstract class, so it's only being written once. This is shown in Appendix A.

The example shows that this kind of code is a nightmare to debug because if the banner suddenly stops scrolling, the first thing that people will notice is that there isn't an "activateAutoscroll" method in the concrete child class implementation. So people will likely add duplicate methods if they're not completely familiar with the class hierarchy.

The code is also not fully unit testable because the view can be directly modified without changing the class state. When the banner stops scrolling, the programmer will not know if it's caused by model's data corruption or the banner view not presenting well.

A better method found was to use a "ViewModel" class to replace the abstract super class in the previous example and replace the hierarchy relationship with a strong pointer. The view controller would always be pointing strongly at its corresponding view model for its entire lifecycle. Strong pointers to objects will prevent them from being deallocated prematurely. This alternative software design pattern is called "MVVM", which stands for "Model-View-ViewModel".

MVVM was created by Microsoft in 2005, derived from its MVC sibling with a few distinct benefits on top. Firstly, it splits the view controller's code into two distinct parts: the view and the view model. Figure 2 shows the component graph for MVVM. [4] The term ReactiveCocoa relates to data binding, which will be explained in later sections.

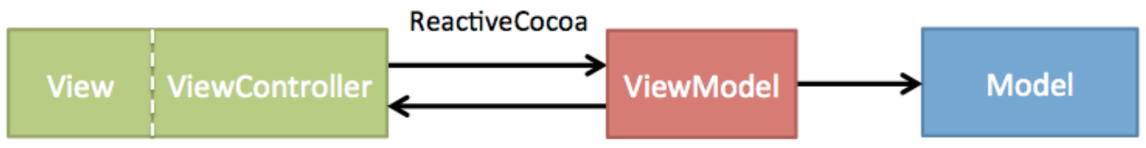

Figure 2: MVVM component graph

Instead of three components like the MVC, MVVM has one more view model class on top of it. According to MVVM, UI related code belongs to views and view controllers and logic code goes into the view model. The view model is completely ready to be unit tested so that data quality can be ensured. Additionally, MVVM further increases modularity and code reusability very much like the class hierarchy did during the first attempt. One interesting observation during coding was that the UI and logic code could be written at the same time by two different developers to increase productivity. The same could not be achieved with the inheritance pattern.

The view model class is a subclass of the generic NSObject, so it cannot easily change any views directly. Instead, it must call predefined methods in the view controller in order to change the view. An example for the view model header file is shown in Appendix B.

When compared to MVC, MVVM needs to have a view model class with every view controller class, so it creates another layer of complexity especially when it comes to data binding the two classes. Perhaps it would be more of a nuisance on smaller iOS applications. Data binding is the process of chaining a view's appearance and the data together. If the user flicks a switch on the touch screen, its corresponding data behind the scenes is automatically changed as well with data binding, and vice versa. The iOS SDK unfortunately doesn't provide an easy way for that to happen. The available options of communication between classes are direct invocation, key value observing, notification center, delegate pattern, and callback blocks. Direct calling would mean all the classes are tightly coupled together, where each class depends on the other ones, which is not an idea option in this case. The other ways for data binding are more or less cumbersome to do, with lots of boilerplate code.

3.0 Implementing MVVM with Ease

The lack of SDK code elegance didn't deter the team from implementing MVVM and enjoy all of the benefits along with it. This is where a functional reactive programming framework for iOS can be really useful.

3.1 Reactive Programming

Reactive Programming is a paradigm focused on data flows and the propagation of change. To be concise, it's the use of asynchronous data streams to achieve automatic, real-time, and global synchronization within a software program. The vocabulary is explained below.

Subscribing to a data streams is like buying stocks. At the moment of purchase, the market value in the personal account is bound to the stock price of the company. This means the personal market value will always relate to the stock price in some way, but the actual numbers don't necessarily have to be equal. On the other hand, data streams can also be unsubscribed, which is analogous to selling stocks. When a stock is sold, the investment is consolidated, and will no longer be affected by other factors. Similarly, when a stream ends, the property or the instance variable that subscribed to the stream will retain the last updated value.

Asynchronous data streams means that the propagation of change is not executed immediately when the method was run, but sometime in the future when some expected change happens.

The data source that the properties subscribe to can come from anywhere, perhaps from a digital thermometer that records the temperature every minute and updates the data model in a software program. With the proper implementation of reactive programming, one can expect the UI to be instantly updated to reflect the change in the model in real-time when the model changes. The best part is that the process is automatic, which requires no extra code. If the change needs to be propagated further to other parts of the program, reactive programming would allow that to happen as well, hence the term "global synchronization".

3.2 Functional Reactive Programming (FRP)

Functional Programming paradigm treats computation as the evaluation of mathematical functions. This means that functions should only be affected by the input variables, but not the current state of the program. When Functional Programming is combined with Reactive Programming, the resulting code can be very concise and powerful.

FRP can do data binding every efficiently, and thus is an excellent companion to the MVVM pattern. Variables can subscribe to a stream of data with as little as one line of code, whereas the delegation and callback patterns require a separate method or block to be defined.

Many frameworks exist in different languages to take advantage of the FRP paradigm. These include Elm and Flapjax. For iOS programming, there is one comprehensive and open-source framework called ReactiveCocoa, or RAC in short. As the name implies, it extends the function of Cocoa Touch to include FRP abilities. One would notice that RAC is very different from other iOS frameworks due to its fluent interface. Instead of invoking methods sequentially, the actions can be linked one after another. This format is more concise than invoking methods separately, but has more or less the same level of code clarity.

The team chose to use RAC because its functionality and amount of documentation far exceeds any other frameworks. It's also open source, so the developers have the power to keep the framework up to date. The quality of the framework is unparalleled with 98 contributors and more than 5,500 commits.

With the use of RAC, iOS developers can easily bind all the view elements with their underlying models within the "viewDidLoad" or "init" method; these methods gets invoked when the view just got loaded on the screen. Developers can effectively use RAC on top of MVVM design pattern to bind data between the view controller and the view model. Appendix C shows a code snippet that shows an example implementation for this binding inside a view controller class. Notice that the

RACObserve macro always takes in a value from the controller's view model. This means the controller always observers the original data from the view model; the view controller itself doesn't need to know anything about the logic side.

Additionally, the RAC framework provides a convenience macro "RAC", which reduce simple data bindings code into one line. The syntax looks very similar to a normal property assignment operation. For an example, consider the code below.

```
RAC(self.appNameLabel, text) = RACObserve(self.viewModel, appName);
```

RACObserve returns a RACSignal object that represents the current and future values of the "self.viewModel.appName" property. RAC then takes the UI property "self.appNameLabel.text" and subscribe that to the RACSignal. A one directional data binding is created as the result.

4.0 Benefits and Drawbacks of using RAC with MVVM

The sections below explain the benefits of using RAC when compared to other methods in various area of interests, which includes code clarity, ease of development, performance, memory management, ease of debugging, and testability.

4.1 Code Clarity

As mentioned before, functional reactive code mostly reside inside the “viewDidLoad” or “init” methods in the very beginning. This allows anyone to easily glance at all the initial setup at once. When compared to KVO or key value observing, RAC is far superior in clarity, even though the functionalities are very much identical.

Key value observing is an informal protocol that defines a common mechanism for observing and notifying state changes between objects. [5] Any object can subscribe to be notified about the state change of a property on another object. However, it requires the developer to implement the following method on each observing class.

```
- (void)observeValueForKeyPath:(NSString *)keyPath
                      ofObject:(id)object
                        change:(NSDictionary *)change
                       context:(void *)context
```

That's just the method name. What's more interesting is that the following method is also required each time an observer needs to be added.

```
- (void)addObserver:(NSObject *)observer
          forKeyPath:(NSString *)keyPath
              options:(NSKeyValueObservingOptions)options
              context:(void *)context
```

To make matters even worse, before the current view controller deallocates, the remove method must be called as shown below, or the application will crash.

```
- (void)removeObserver:(NSObject *)anObserver forKeyPath:(NSString *)keyPath
```

Clearly, KVO has a lot more boilerplate code than RAC, making the code less readable. Appendix D compares KVO to RAC with a simple example. Although simple, the actual difference during implementation is night and day. The KVO used 26 lines of code to observe a single property with lots of ugly casting and calling to the "isKindOfClass" method. Meanwhile, RAC used a mere 4 lines to do the same thing.

For the other three methods of inter-class communication: NSNotificationCenter, Delegate, and callback blocks, the difference is even bigger. One would need to write so much code that it's next to impossible to setup a pair of sending and receiving mechanism for each property being observed.

In terms of code clarity, choosing RAC was an obvious choice for data binding.

4.2 Ease of Development

It takes a quite a while for a developer to fully grasp the concept of FRP. The syntax is very different and each of the components in RAC serves a specific purpose. It's very easy to confuse a method meant for one object with another because they all look so similar. Since RAC is a well-maintained framework, developer will tend to find the method they are looking for most of the time. Search for the right method to use, however, is not always an easy task in the beginning. After the steep learning curve, the development process becomes much more comfortable but not without caveats.

Once, the team discovered RAC code in two different classes that contains two separate instances of the PBActiveWatch signal object. When one of the classes got the signal that the active watch has been connected, it immediately calls for UI to switch to the next screen. Upon loading, the next view controller looks at a different instance of PBActiveWatch and discovers that the second one is still empty. The signal was then unified into one instance to resolve the problem.

Other times, the object that the developer is observing might have changed, but not in a way that was expected. So extra efforts must be taken to filter out the unwanted values. For example, when a PBWatch instance transitions from empty (nil) to a valid object, a firmware update check is performed. However, the version info inside the watch object must also be valid for it to work. The initial code did not account for that scenario, and triggered the firmware update check too early.

Overall, most people seem to like the new fluent interface imposed by RAC. It eliminates the use of intermediary variables, and makes it easy to write code. The time needed to master RAC, however, is very long.

4.3 Performance

ReactiveCocoa involves the flow of data in the background, which in most cases should not pose a threat to the responsiveness of the app. However, if the code is not written properly, one might encounter subtle bugs. During the development, the team has encountered a RACSignal that constantly fires. The extra processing takes up processing time and slows down the app. To prevent such cases from ever happening, the throttle method: "- (RACSignal *)throttle:(NSTimeInterval)timeInterval" was used to define a minimum time interval before the signal can fire again. The team didn't notice any other performance differences between RAC and other type of mechanisms.

4.4 Memory Management

Next, the developers compared memory management between standard Cocoa framework APIs (Application Programming Interfaces) and the ReactiveCocoa interface. API in this case is a set of public methods that can be used to develop a piece of software.

For the majority of time when the developers are using standard workflows within RAC, memory management can be easily set up when the RACSignal is being created. In some instances where the developer knows that the view controller will exist for the entire duration of the application, no memory management for that RACSignal is required.

RAC simplified the code a lot during the development process because the developer didn't need to manually retain every RACSignal. Instead, the developer simply subscribes to a RACSignal and a subscriber object is automatically created by RAC behind the scenes. RAC keeps the same number of retain counts as the number of subscriptions to the RACSignal. When there're no more subscriptions left, the subscriber object will be automatically deallocated on the next run loop. This feature virtually eliminates the use of properties and state, which certainly is a welcoming surprise when compared to the standard development flow.

For instances where RACSignals should not persist for the whole duration of the application, the team found those to be very elegant to deal with. There're three ways that this was achieved. The easiest one is shown in Appendix E as an example.

The example has the "- (RACSignal *)takeUntil: (RACSignal *)signal" method that removes the subscriber when the current view controller gets deallocated. Specifically, the method of interest is:

"takeUntil:self.rac_willDeallocSignal".

This is part of RAC's fluent interface so that it doesn't hinder the flow when programming.

The second way is using the method "- (RACSignal *)take: (NSUInteger)count". Similar to the previous method, this one takes in an unsigned integer specifying the number of changes that the subscriber listens to before it gets deallocated automatically.

The third way was to keep the RACDisposable object that a subscribe method returns, as shown in Appendix F. RACDisposable is an object used to dispose a subscriber manually, giving developers the power to do memory management. The main idea here is to have RACDisposables from each subscription to look up to one main RACDisposable, and dispose the main RACDisposable when the view controller deallocates.

We found that the first two ways are very easy to do, and thus those were used extensively throughout the Pebble iOS app.

4.5 Debugging

The debugging process for code that's written with RAC is difficult. Normally, Xcode provides a convenient feature to allow a developer create a breakpoint for all exceptions that happens. If the application ever crashes, this will allow developers to track down the exact line of code that causes the problem on the call stack.

Unfortunately, RAC updates properties automatically through whenever properties change, therefore Xcode cannot show the source of the problem. Upon the creation of an erroneous signal, the program executes as usual. It's when the problematic signal fires that causes the program to crash. At that point, the developer does not get any useful debug information in the Xcode debug navigator as shown in figure 3, nor a useful debug message in console as shown in Appendix G.

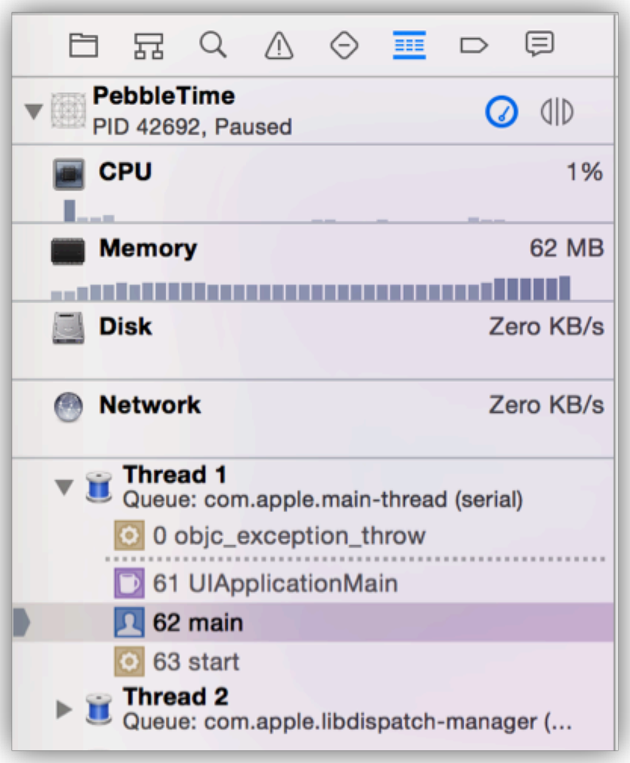

Figure 3: Xcode debug navigator at the time of program crash

4.6 Unit Testing

At Pebble, the iOS team uses the Kiwi testing framework instead of the default XCTesting. There really isn't much difference between the two unit testing frameworks. Perhaps Kiwi was initially chosen for its block syntax, which looks slightly more organized than the standard method syntax. A code snippet written with Kiwi is provided in Appendix H.

When unit testing the classes, the team made sure that the test file only have access to the private header (.h) file of the class. In circumstances where a private method do need to be exposed to testing, a separate "+testing.h" category must be created, so that normal application code will not have access to those methods. A category in Objective-C extends the functionality of a class without subclassing, so it's a low maintenance and modular alternative when a subclass is not absolutely necessary.

With these limitations in mind, one would realize that testing code written with RAC is basically the same as non-RAC code since the testing file doesn't need to care about the specific implementation details. However, if the methods being tested take in RAC related objects as their parameters or returns RAC objects, then extra testing code is required to decompose the RAC objects into standard Objective-C objects. The little extra work needed to test RAC is hardly noticeable compared to its benefits.

5.0 Conclusion

Adopting the MVVM design pattern is a great choice for big software projects with multiple developers as it shrinks the code size for view controller classes. However, for smaller projects, it’s not as appealing because for every view controller class, there also needs to be a view model associated with it.

The MVVM pattern depends on the concept of data binding, and should be used in conjunction with RAC. There are lots of benefits for using RAC such as code clarity and ease of development for experience programmers. On the other hand, there are drawbacks such as a very steep learning curve in addition to some extra code to take care of possible memory leaks and performance issues.

The steep learning curve will prove to be challenging for new employees and interns alike, but it’s very well worth the effort to understand RAC and use it to simplify code. There are lots of methods available to use, and they cover almost all the use cases.

When creating data bindings in MVVM, it’s best for developers to use RAC instead KVO. In other scenarios, RAC can also be used for its elegance.

Debugging with RAC code is often a challenge. One will spend more time than usual fixing bugs that crashes the application. This is one of the prices to pay for getting automatic signaling instead of manual message passing.

The choice of adopting MVVM and RAC has saved lots of time, and has proven to be invaluable to the iOS project. Pebble should continue use the two frameworks to cut down on development time and increase code testability.

6.0 Recommendations

There are lots of things that can be simplified further with the induction of the new Swift language in 2014. It provides lots of good features such as Tuple structure for returning multiple values, type defined arrays and dictionaries, and guaranteeing the existence of an object if needed. In addition, Swift is fully compatible with Objective-C. Developers at Pebble should explore the possibility of using Swift along with MVVM and RAC and see if it would help the development process even further.

One blog post from iOS team at SoundCloud has claimed that they have used an open source tool that would allow the developer to know the source of the problem when using RAC.

> *"Backtraces tend to get longer but this can be alleviated with some extra tooling like custom LLDB filters."* [6]

It's possible to reduce the amount of time spent on debugging by implementing the tool like SoundCloud did. This option should also be explored to make RAC debugging experience better.

During the previous four-month sprint towards the launch of Pebble iOS 3.0 app, the team has cut corners on some of the unit testing. The testing should resume once the app is made available to the public to ensure continued code quality.

7.0 References

1. Pebble Technology, (2012, April 11). *Pebble: E-Paper Watch for iPhone and Android*. Retrieved May 1, 2015, from Kickstarter: https://www.kickstarter.com/projects/597507018/pebble-e-paper-watch-for-iphone-and-android?ref=most_funded

2. Pebble Technology, (2012, April 11). *Pebble: E-Paper Watch for iPhone and Android*. Retrieved May 1, 2015, from Kickstarter: https://www.kickstarter.com/projects/597507018/pebble-time-awesome-smartwatch-no-compromises?ref=most_funded

3. Reenskaug, T. (2003, August 20). *The Model-View-Controller (MVC) Its Past and Present.* Retrieved May 1, 2015, from University of Oslo: http://heim.ifi.uio.no/~trygver/2003/javazone-jaoo/MVC_pattern.pdf

4. Eberhardt, C. (2014, July 3). *MVVM Tutorial with ReactiveCocoa: Part 1/2*. Retrieved May 1, 2015, from Ray Wenderlich: http://www.raywenderlich.com/74106/mvvm-tutorial-with-reactivecocoa-part-1

5. Thompson, M. (2013, October 7). *Key-Value Observing*. Retrieved May 1, 2015, from NSHipster: http://nshipster.com/key-value-observing/

6. Sezgin, M. (2014, July 7). *Building the new SoundCloud iOS application — Part I: The reactive paradigm*. Retrieved May 1, 2015, from SoundCloud: https://developers.soundcloud.com/blog/building-the-new-ios-app-a-new-paradigm

Appendices

Appendix A: View controller inheritance example

Parent Class: JPMainExploreViewController

```
- (void)viewWillAppear:(BOOL)animated
{
  [super viewWillAppear:animated];
  [self.bannerView activateAutoscroll];
}
```

Child Class: JPiPadExploreViewController

```
- (void)viewWillAppear:(BOOL)animated
{
  [super viewWillAppear:animated];
  // Do nothing
}
```

Appendix B: Example view model header file

```
//
//  PBDetailPopupViewModel.h
//  PebbleApp
//
//  Created by Si Te Feng on 2/20/15.
//  Copyright (c) 2015 Pebble Technology. All rights reserved.
//

@import Foundation;

#import "PBProvider.h"
#import "PBWatchApp.h"

@class PBCurrentLockerAppManagerProvider;
@class PBDetailPopupAction;
@class PBDetailPopupTableCell;
@class PBDetailPopupViewModel;
@class PBTimelineManager;
@class PBWatchApp;

@protocol PBDetailPopupViewModelDelegate <NSObject>

- (void)viewModel:(PBDetailPopupViewModel *)viewModel
openSettingsForWatchApp:(PBWatchApp *)watchApp;
- (void)openCalendarSettings:(PBDetailPopupViewModel *)viewModel;
- (void)openWeatherSettings:(PBDetailPopupViewModel *)viewModel;

- (void)showCompanionAppForViewModel:(PBDetailPopupViewModel
*)viewModel;

-
(void)presentAppAboutWebPageForViewModel:(PBDetailPopupViewModel
*)viewModel;
- (void)presentEmailDeveloperForViewModel:(PBDetailPopupViewModel
*)viewModel;

- (void)viewModel:(PBDetailPopupViewModel *)viewModel
openShareMenuWithWatchApp:(PBWatchApp *)watchApp;

@end


@interface PBDetailPopupViewModel : NSObject

@property (nonatomic, weak) id<PBDetailPopupViewModelDelegate>
delegate;

@property (nonatomic, assign, readonly) BOOL showIcon;
@property (nonatomic, assign, readonly) BOOL showScreenshot;
@property (nonatomic, assign, readonly) BOOL showQuickGlance;
@property (nonatomic, assign, readonly) BOOL
showGetCompanionButton;

@property (nonatomic, strong, readonly) NSString *appName;
@property (nonatomic, strong, readonly) NSString
*appDeveloperName;

// If appIconImage is set, it will be used instead of the default
appIconURL.
@property (nonatomic, strong, readonly) NSURL    *appIconURL;
@property (nonatomic, strong, readonly) UIImage  *appIconImage;

// If appScreenshotImage is set, it will be used instead of the
default appScreenshotURL.
@property (nonatomic, strong, readonly) NSURL
*appScreenshotURL;
@property (nonatomic, strong, readonly) UIImage
*appScreenshotImage;

@property (nonatomic, strong, readonly) NSURL
*appCompanionURL;
@property (nonatomic, copy, readonly) NSString   *numberOfHearts;

@property (nonatomic, copy, readonly) NSString   *developerEmail;
@property (nonatomic, strong, readonly) NSString *appId;

@property (nonatomic, assign, readonly) BOOL hasMetaInformation;
@property (nonatomic, assign, readonly) BOOL hasSettings;
```

```
@property (nonatomic, assign, readonly) BOOL canBeDeleted;
@property (nonatomic, assign, readonly) BOOL
canBeSetAsDefaultWatchFace;
@property (nonatomic, assign, readonly) BOOL usesTimeline;
@property (nonatomic, assign, readonly) BOOL
areTimelinePinsEnabled;
@property (nonatomic, assign, readonly) BOOL
areTimelineNotificationsEnabled;

+ (instancetype)viewModelWithWatchApp:(PBWatchApp *)watchApp
lockerAppManagerProvider:(PBCurrentLockerAppManagerProvider
*)lockerAppManagerProvider timelineManager:(PBTimelineManager
*)timelineManager;

- (void)openSettings;
- (void)deleteWatchApp;
- (void)setAsDefaultWatchFace;
- (void)getCompanionApp;

- (void)toggleAreTimelinePinsEnabled;
- (void)toggleAreTimelineNotificationsEnabled;

- (void)openAboutWatchApp;
- (void)openContactDeveloper;
- (void)share;

@end
```

Appendix C: PBDetailPopupViewController.m viewDidLoad method implementation

```
- (void)viewDidLoad {
  [super viewDidLoad];

  RAC(self.appNameLabel, text) = RACObserve(self.viewModel,
appName);
  RAC(self.appCompanyLabel, text) = RACObserve(self.viewModel,
appDeveloperName);

  @weakify(self);
  [[RACSignal
   combineLatest:@[RACObserve(self.viewModel, appScreenshotURL),
RACObserve(self.viewModel, appScreenshotImage)]]
   subscribeNext:^(RACTuple *URLAndImage) {
     @strongify(self);
     RACTupleUnpack(NSURL *URL, UIImage *image) = URLAndImage;

     if (image) {
       [self.appScreenshotView setImage:image];
     } else {
       [self.appScreenshotView setImageWithURL:URL
placeholderImage:PBImagesCatalog.listPlaceholderIconImage];
     }
   }];

  [[RACSignal
    combineLatest:@[RACObserve(self.viewModel, appIconURL),
RACObserve(self.viewModel, appIconImage)]]
   subscribeNext:^(RACTuple *URLAndImage) {
     @strongify(self);
     RACTupleUnpack(NSURL *URL, UIImage *image) = URLAndImage;

     if (image) {
       [self.appIconView setImage:image];
     } else {
       [self.appIconView setImageWithURL:URL
placeholderImage:PBImagesCatalog.listPlaceholderIconImage];
     }
   }];

  [self.favLabel rac_liftSelector:@selector(setupWithImage:text:)
             withSignalsFromArray:@[[RACSignal
return:PBImagesCatalog.heartIconImage],
                                    RACObserve(self.viewModel,
numberOfHearts)]];
}
```

Appendix D: KVO versus RAC comparison for observing a single property

Key Value Observing method

```
static void * const MyClassKVOContext =
(void*)&MyClassKVOContext;

- (void)viewDidLoad {
  [super viewDidLoad];

  [self addObserver:self forKeyPath keypath(self.viewModel,
customLabelString) options:NSKeyValueObservingOptionNew
context:MyClassKVOContext];
}

- (void)observeValueForKeyPath:(NSString *)keyPath
                      ofObject:(id)object
                        change:(NSDictionary *)change
                       context:(void *)context
{
  if ([object isKindOfClass:[NSString class]]) {
    if (context == &MyClassKVOContext) {
      self.customLabel.text = (NSString *)object;
    }
  } else {
    [super observeValueForKeyPath:keyPath ofObject:object
change:change context:context];
  }
}

- (void)dealloc {
  [self removeObserver:self forKeyPath:keypath(self.viewModel,
customLabelString) context:MyClassKVOContext];
  [super dealloc];
}
```

ReactiveCocoa method

```
- (void)viewDidLoad {
  [super viewDidLoad];
  RAC(self.customLabel, text) = RACObserve(self.viewModel,
customLabelString);
}
```

Appendix E: Setup subscription to be deallocated automatically

Inside PBFirmwareUpdateViewController.m:

```
RAC(self, watchIsInRecoveryMode, @NO) =
      [[[[RACObserve(self, watch)
       flattenMap:^(PBWatch *watch) {
         return watch.versionInfoSignal;
       }]
       map:^(PBVersionInfo *versionInfo) {
         return
@(versionInfo.runningFirmwareMetadata.isRecoveryFirmware);
       }]
       takeUntil:self.rac_willDeallocSignal]
       catchTo:[RACSignal return:@NO]];
```

Appendix F: Deallocating observers method 3

Within @interface:

```
@property (nonatomic, strong) RACDisposable *disposable;
```

Within @implementation:

```
- (instancetype)init {
    self.disposable = [[signal subscribeNext:^{
      /* do stuff */
    }] asScopedDisposable];
}

- (void)dealloc {
  [self.disposable dispose];
  [super dealloc];
}
```

Appendix G: Example debug string print out to the console

```
2015/05/07 18:11:23:767-04:00 <NSLog> *** Terminating app due to
uncaught exception 'NSInvalidArgumentException', reason: '-
[__NSCFNumber length]: unrecognized selector sent to instance
0x7beb5940'
```

Appendix H: Example Kiwi testing code for PBAppRegistryBlob

```
//
//  PBAppRegistryBlobTests.m
//  PebbleApp
//
//  Created by Si Te Feng on 2/4/15.
//  Copyright (c) 2015 Pebble Technology. All rights reserved.
//

#import <Kiwi/kiwi.h>
#import "PBAppRegistryBlob.h"
#import "PBVersionNumber.h"

SPEC_BEGIN(PBAppRegistryBlobTests)

__block NSString *appName;
__block NSUUID *uuid;
__block PBProcessInfoFlags infoFlags;
__block PBIconResourceIdentifier iconResourceID;
__block PBVersionNumber *appVersion;
__block PBVersionNumber *sdkVersion;

beforeAll(^{
  appName = @"Test App";
  uuid = [[NSUUID alloc] initWithUUIDString:@"68753A44-4D6F-1226-
9C60-0050E4C00067"];
  infoFlags = 0x12341234;
  iconResourceID = 0x12343271;
  appVersion = [[PBVersionNumber alloc] initWithMajorVersion:0x32
minorVersion:0x92];
  sdkVersion = [[PBVersionNumber alloc] initWithMajorVersion:0x52
minorVersion:0x50];
});

describe(@"Initialization", ^{

  it(@"Should store small values accurately", ^{
    PBAppRegistryBlob *app = [[PBAppRegistryBlob alloc]
initWithAppName:appName uuid:uuid infoFlags:infoFlags
iconResourceID:iconResourceID appVersion:appVersion
sdkVersion:sdkVersion];

    [[app.appName should] equal:appName];
    [[app.uuid should] equal:uuid];
    [[theValue(app.flags) should] equal:theValue(infoFlags)];
    [[theValue(app.iconResourceID) should]
equal:theValue(iconResourceID)];
    [[theValue(app.appVersion) should]
equal:theValue(appVersion)];
    [[theValue(app.sdkVersion) should]
equal:theValue(sdkVersion)];
  });
});

SPEC_END
```