

Technology Scholarship

Introduction	3
Issue	3
Initial Meeting	3
Stakeholders	5
Research	5
<i>The Current Solution</i>	5
<i>The Advantages of an App-based Solution</i>	5
<i>Existing Products</i>	6
<i>Exercise Books - Design Within Another Field</i>	7
<i>Development Tools</i>	9
Client Change	11
Planning for Testing	12
Design Mockup	12
Research	15
<i>Core Data</i>	15
<i>Contacts Integration</i>	15
<i>Camera Integration</i>	16
<i>iOS 7 UI Concepts</i>	16
<i>Script Evaluation</i>	17
<i>Password Security</i>	18
Stakeholder Input	18
Final Brief	19
<i>Conceptual Statement</i>	19
<i>Specifications</i>	19
Prototype Development	20
Client Meeting	20
Stages of Development for the Initial Screens	21

The Splash Screen	21
Style Guide	23
Localisation	23
Database Planning	24
Implementing Encryption	25
The Client Selection Screen	25
<i>Contacts Integration</i>	25
<i>Writing Convenience Methods for Getting Person Info</i>	26
<i>Implementing Search</i>	28
<i>Proper Encapsulation, or, "So this is how model objects are supposed to be used"</i>	29
<i>The Functional Client Selection View</i>	30
The Health Wheel	31
<i>Going Back to the Client List</i>	34
The Questionnaire	35
Refactoring	38
Importing Exercises	38
Scripting for Result Conversion	39
<i>Accuracy of Algorithms</i>	40
The Exercise Screen	41
<i>User Considerations</i>	41
User Customisation	42
Final Client Meeting	43
Final Evaluation	45
<i>Reflection</i>	46
<i>Conclusion</i>	47

Introduction

I was approached by my father's personal trainer about the possibility of making the first version of a mobile application (an 'app') for use with his clients. The personal trainer and a group of others were starting a new company, and they wanted a way to be able to track and store their clients' progress on their devices.


At the time, the client had already approached another developer, who had produced basic wireframes and specifications for an application. However, the client was dissatisfied with the concepts and the potential quality of the application, and chose to continue with a different developer.

I decided to undertake development on this because I felt that building a professional app for a client would allow me to progress and develop my design and coding skills. The context of health and fitness training is one which I was unfamiliar with, so it was also an opportunity to learn more about the role and needs of a personal trainer.



Issue

There are no existing apps that allow personal trainers to easily and effectively show their clients their holistic health and progress.

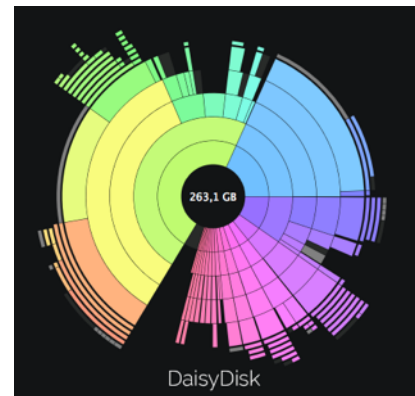
In addition, most similar apps are user focused, rather than personal trainer focused, which makes it difficult for trainers to keep track of their clients' progress and develop programmes to suit their needs. In particular, existing apps can only process the exercise results from a single person, meaning that for use with a trainer, only one of their clients could be tracked, or else each client would need a copy of the app. There is therefore a lack of trainer focused apps. 

Initial Meeting

I organised a meeting with the client to discuss their expectations for the app's functionality. From this meeting I listed out a few key features which would be the key components of the app.

- The app would be for the use of the personal trainers, to display information to their clients.
- The app needs to keep track of clients and store individual data for each client.
 - The app would integrate with the Contacts app to store their details and personal information.
 - My client's company requires each of their clients to complete a questionnaire. This needs to be able to be retrieved on request, and some data (e.g. gender, age) would be used in-app for calculations.
- The app's data would be displayed in a 'performance wheel.' This wheel would visually represent the client's progress in a number of different trials.

- When my client described the concept of a performance wheel, I immediately thought of a program I have used named [DaisyDisk](http://www.daisydiskapp.com) (www.daisydiskapp.com). The visual style of that program sounded very similar to what the client was describing.
- The app needs to have a database of exercises. This data would be provided by the client. Each exercise would have a formula to convert data from their client's test to a number for the performance wheel; the client's results would therefore be stored in the app.
- The app would need to hold identifiable data and keep it secure.
- The app needs to be able to send clients a report of their progress or exercises through email.



When you first open the app, you are greeted with a list of your clients in a table view. You can tap a button in the top right to add a new client, in which case a selection box would slide up from the bottom of the screen asking whether you wish to create a new client or load one from your Contacts. Following this, you would be able to enter basic information about this client. Following this step, the client would fill out their initial questionnaire - the app's functions would be restricted until the client had completed the questionnaire. They would then be added to the list.

Search functionality could be added to the list of clients if necessary - else, they would be sorted alphabetically.

Once a client has been selected, the client list would slide away to reveal a screen with three tabs at the bottom. The first tab contains the visual 'performance wheel', along with some basic details about the client - for example, age, gender and height. In the top right hand corner there is a button which lets you change the selected client, while in the top left there is a button which allows you to edit this client's details.



The second tab contains the results of the initial questionnaire. The contents of this tab cannot be changed, and are for viewing only.

The third tab would be a list of workouts, also in table view form. You could create a new workout by tapping on the + button in the top right hand corner, and a screen would displaying allowing you to choose 12 (or however many) of the exercises on offer. Once a workout has been created and selected, a list with the selected exercises would display. Each exercise could be tapped on, and then the statistics for that exercise could be entered or, if already entered, displayed. This screen would also display a graph or numbers which inform the trainer of how the client has progressed on that particular exercise, if relevant.

- Initial, rough ideas of how the app could work. I noted this down after my first meeting with my client, and sent it to my client as a record.

Initially, the client wanted compatibility across as many devices as possible, including a version for Google's Android operating system. However, I explained, giving examples of popular apps, how trying to make applications for multiple devices usually results in a product that feels at home on none of them. I also estimated that it would lengthen development time if I did a platform-specific port for Android, for example. I did give the option of trying to avoid iOS-specific APIs in order to make a future porting effort easier. The context consideration of 'native-feeling' apps clashed with the client's desires. When I explained this to the client, he responded with certainty that he would therefore prefer to focus on a single platform.




Stakeholders

Primary Stakeholders: The primary stakeholders will be my client and the other personal trainers at his company. They are the end users, and will be the only ones who, at least in the initial concept, would input data into the app. The app therefore needs to be primarily tailored to their needs in terms of usability.

Secondary Stakeholders: The primary stakeholders would use this app with their own clients, who are the main secondary stakeholders. They would receive the output of the app, both through email and visually. Therefore visual considerations are the main component they would affect. It is possible that their needs might clash with the trainers' needs; for example, a particular person might prefer that their file can only be unlocked by a password known to them, whereas the trainer might wish to look at that file at other times to plan. In cases such as these, the needs of my client, the trainer, come first.

The gym is also a wider community stakeholder. My client intends to use this app to add value to his sessions with clients, which may drive business to the gym. The exercises included in the app could also affect demand for certain equipment: for example, the equipment required to complete the exercises within the app would be under greater demand than the equipment that is not needed.

If my client were to release the app on the App Store, Apple would also be a stakeholder. They receive a 30% portion of the profits from any sales, and they also keep records of statistics that can affect an application's ranking in the App Store charts .

Research

The Current Solution

I asked my client to compare the potential solution with the current method of carrying out assessments, and I have quoted his response below.

Currently trainers assess their clients in an inefficient manner. The process involves reading up on the correct assessment for the client in question, carrying out the assessment and recording the info either on a device or physically with pen and paper. An issue with this is that the trainer then has to input the data 'after hours,' which is often a lengthy process. This creates less incentive to measure and track a clients' progress, meaning that the client and trainer will have less of an idea if the training is doing what it is supposed to do. The client also does not get results displayed properly and if they did this would require even more background time.

There are no advantages to assessing a clients' progress with current methods.

The Advantages of an App-based Solution

I then asked him to say what he thought some of the primary advantages of the proposed solution would be.

Trainers would still have to ensure that they're using the correct assessment for a particular client; however, the assessments will be categorised, meaning they're better able to make the correct choice with basic training. The trainer is able to carry out the test

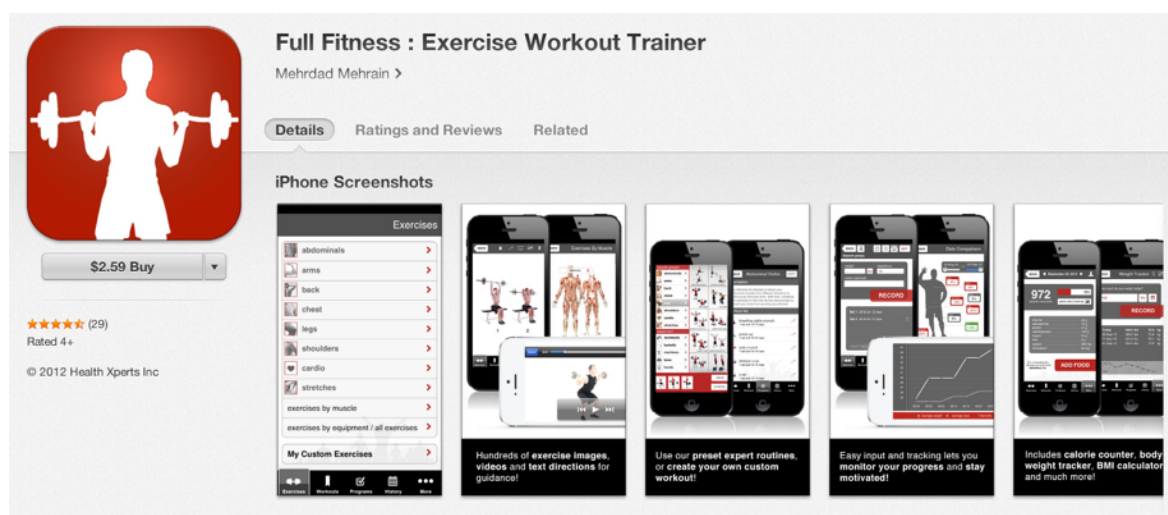
efficiently and effectively in 'real time' meaning that the client has a well presented report that shows progression and tracked progress by the time they able to open their inbox. This is a major advantage in two ways:

1. The trainer spends little to no administrative time inputting data while providing a much higher level of service and knowing that what they're doing with a client is working.
2. Their client is reassured that their best interests are looked after and that the time, money and effort they're spending with a trainer is moving closer to what is required or what is desired depending on the situation.

Existing Products

With the initial requirements in mind, I started to research what was available in other apps. My target platform would be the iOS operating system, so my context was health/fitness training apps for the iOS platform. Given the App Store has a fitness category, I looked at the most popular apps and tried to analyse what had made them popular.

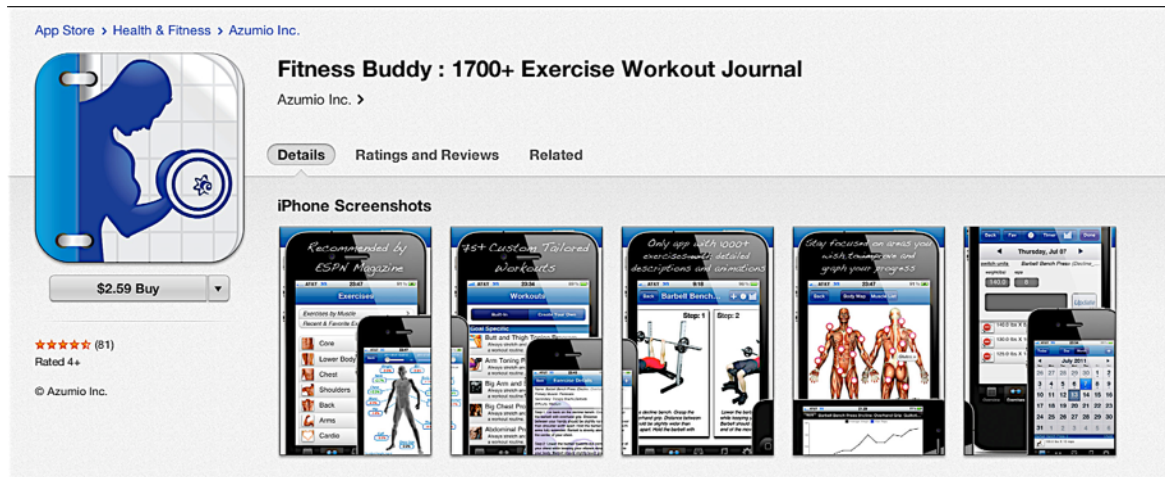
Most of the apps are targeted at the secondary client, so there was little precedent for features targeted directly at personal trainers.



The most popular app in the category when I looked was "Full Fitness: Exercise Workout Trainer". I found a review of the app at <http://iphone.appstorm.net/>, and noted down their thoughts. The review started with the aesthetic appeal: the app uses a standard navigation controller and tab bar controller layout, something that I've used in my own apps, but has embellished it with "nice use of a red and grey colour scheme" - bold and consistent colours. The reviewer pointed out the attractiveness of graphs to visually display data. I also noted the background silhouettes, which added visual flair to the grey background.

The reviewer liked the comprehensiveness of the database, its helpful images and detailed instructions. The app includes the ability to track user progress as well, similar to my client's app; however, while the reviewer appreciated the feature, they said that for the calorie counter in particular the user interface (UI) could be a hinderance.

I also looked at “Fitness Buddy: 1700+ Exercise Workout Journal”.



All Things Digital did a [review](http://allthingsd.com/20120802/fitness-apps-turn-your-iphone-into-a-personal-trainer/) (allthingsd.com/20120802/fitness-apps-turn-your-iphone-into-a-personal-trainer/) of both “Fitness Buddy” and “Full Fitness”’s predecessor. They discussed how the high number of exercises in “Fitness Buddy” could be daunting, but in my case, the exercises are the decision of my client, rather than mine. They appeared to take for granted the ability to log progress, whereas features such as custom music playlists drew mention. Their main complaints centred around the UI; they thought that although “There are icons located at the top of each page, it’s not immediately clear what they do.” It seems that clean UI is a major differentiating feature, and so UI design should be a major focus for me in making the app, probably more than quantity of features.

Exercise Books – Design Within Another Field

I also wanted to take a look at how other technologists had presented exercises so that they were clear and easy to follow. To do this, I looked at Adam Campbell’s “Big Book of Exercises” series, wherein he presents a range of different exercises for people to train off. Judging by the four and a half star Amazon ratings for both the [men’s edition](http://www.amazon.com/The-Mens-Health-Book-Exercises/dp/1605295507) (www.amazon.com/The-Mens-Health-Book-Exercises/dp/1605295507) and the [women’s edition](http://www.amazon.com/The-Womens-Health-Book-Exercises/dp/B00CC6DG7O) (www.amazon.com/The-Womens-Health-Book-Exercises/dp/B00CC6DG7O), people generally found these books effective and useful. Note that embedded images were obtained from Amazon’s Look Inside function.

Chest | PUSHUPS

In this chapter, you'll find 63 exercises that target the muscles of your chest. Throughout, you'll notice that certain exercises have been designated as a Main Move. Master this basic version of a movement, and you'll be able to do all of its variations with flawless form.

PUSHUPS AND DIPS

These exercises target your pectoralis major. However, they also hit your front deltoids and triceps, since these muscles assist in just about every version of the movements. What's more, your rotator, trapezius, serratus anterior, and abdominals all contract to keep your shoulders, core, and hips stable as you perform the moves.

MAIN MOVE Pushup

- Get down on all fours and place your hands on the floor so that they're slightly wider than and in line with your shoulders.



In the main descriptor for an exercise, there is an image of a person performing that exercise; my client will provide these images for me. The primary instruction distinguishes itself from the detailed instructions through its serif font and light grey font colour, in contrast to the detailed instructions' blue-green sans-serif font. This clearly differentiates the text, placing priority on the more important information through its 'heavier' typesetting (the serifs give the font extra 'weight') and the darker colour.

The image is set against a clean white background, again giving it clear visual distinction. Each detailed instruction has an arrow (coloured in the text's complementary colour) pointing to the relevant part of the image. The arrows are curved in such a way as to reinforce the text's message; for example, the instruction saying "your arms should be straight" is accompanied by an arrow pointing vertically down, suggesting that the arms' force on the ground should be a continuation of that arrow. Likewise, the instruction to "brace your abdominals as if you were about to be punched in the gut" has an arrow which indicates the direction of that punch.



The style is continued in the variations, with the addition of stylised information boxes. These use the font and colour of the more important text from the initial page, but lighten its impact through the use of italics and a light blue outline.

Throughout both of these pages, a clear, clean, and spacious visual style is utilised, which helps to communicate the message through to the reader. I should aim to emulate this aspect in my app.

Development Tools

There are many tools, frameworks, and programming languages available to develop iOS apps, each with its own advantages and disadvantages.

Xcode

Xcode is the default IDE (integrated development environment) used to develop iOS apps. It has built-in tools for storyboarding your app and submitting it to the app store. I have prior experience with it, having made an iOS app using it before.

Xcode supports many different programming languages, with the primary languages for iOS development being Objective-C and C. All the native frameworks and UI elements are provided via Objective-C or C APIs, so those



languages, along with C++, are often used for making iOS-only programs. Furthermore, Xcode needs to be used for the final deployment of any iOS app.

Objective-C is an object-oriented language, and the APIs strongly support a Model View Controller coding pattern, which I have found in the past to be useful for compartmentalised, maintainable code. The language conventions favour verbosity, which means that Objective-C code is often highly readable. This is especially important as I may not be the person continuing work on it after the initial version is released – should my client choose to develop later versions with a different developer, that developer would need to be able to understand and modify the code. However, it is more low-level than many scripting languages, and in code comparisons has been shown to require more code to accomplish the same task, which may lengthen development – this is undesirable, as my client wants the app completed as soon as possible.

Xcode is a free download and has no fee other than the NZ\$139/year required to release any app for iOS.



Corona SDK

The Corona SDK (www.coronalabs.com/products/corona-sdk/) is a toolkit for developing mobile applications. It can produce application binaries that run on both iOS and Android. This has a major benefit over making an app with Xcode, wherein an app would have to be almost completely rewritten to port it to a different platform. However, this also has the result that non-native UI elements are used and platform conventions aren't as encouraged by the APIs; it is therefore more suited to games, which are expected to have completely custom interfaces.

Corona uses Lua as its programming language. Lua is a diverse, high-level scripting language which can be adapted for many different programming paradigms. As it is interpreted rather than compiled, it does have a performance deficit compared to many compiled languages when performing identical tasks. In addition, its dynamism means that it can encourage a looser structure, causing issues for code readability.

I have little experience with Lua or the Corona APIs, so using them could mean a longer development time.

Corona starts at no cost, but costs US\$600/yr to interface with the native APIs directly or for such features as having a custom splash screen.

Xamarin

Xamarin (xamarin.com) is a suite of cross-platform APIs and compilers. It has a greater emphasis on platform-specific UI code than Corona; each platform it supports has links into the native APIs. This means that for developing an iOS app I would be the UIKit framework for both Objective-C and Xamarin development: the only difference would be the language, which is C# for Xamarin. I have some experience with C# in making other apps.

Xamarin highlights on their site the decreased amount of code (albeit in carefully selected examples) necessary compared to Objective-C. However, as C# is a scripting language, there would also be a performance cost compared to compiled code.

Although there would be more reusable code for porting to another language with Xamarin, it still seems probable that it would simply be a wrapper around the native Objective-C APIs.

Xamarin is free for individuals, but has a restricted maximum app size. The cost for use by an organisation is US\$999.



Conclusion

Having assessed these options, I decided to use Xcode for development. My main priority was cost, and neither Xamarin nor Corona offered, at a reasonable cost, a version without restrictions that would limit the app's value to my stakeholders. In addition, Xcode was the tool I was most familiar with, and would allow me to focus on learning more advanced technologies and features. The speed gains from me being familiar with it would outweigh any time cost of having to write slightly more code. Objective-C's verbose syntax means that code readability is enhanced to the point of well-written code being almost self-documenting. The other two SDKs' cross-platform advantages were not enough to outweigh Xcode's benefits for me, and, as my client did not express cross-platform compatibility as a priority, I feel comfortable in my decision to use it.

Client Change

A while after my initial meeting and subsequent followup with the client, I learned that he had decided to step away from his company, and that I would be dealing with a new client. This gave me an opportunity to get feedback from a second personal trainer as to what they might want from the app.

The new client was aware of what had been discussed, and mostly wanted the same things.

The feature additions he wanted were:

- The ability to take and store five second videos for certain assessments as a record.
- The ability to take photos and link them to certain assessments
- The ability for the user to incorporate their own branding into the app; in other words, the ability for the application to be modified and adapted for different companies to suit their needs.

He reinforced the importance of the health wheel method of graphical display, and also informed me that it should be the app's role to not only track the results of the exercises but also to actually store the information on how to perform them. This reflected the shift in thinking to a more general product that could be used by not only personal trainers within his company but also other personal trainers.



However, in general, his feedback was very much in line with what I was planning. This gave me confidence in the planned solution.

Planning for Testing

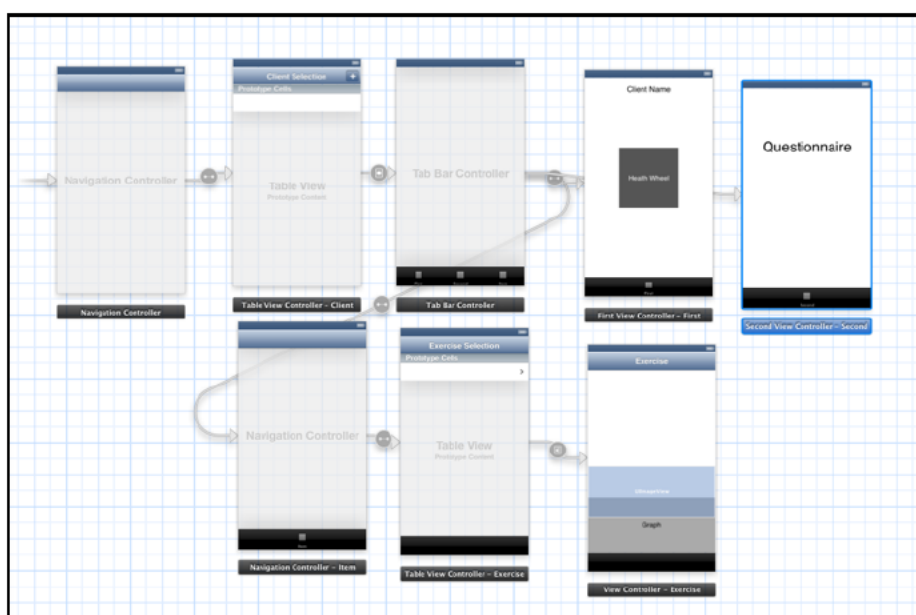


This application is going to be fairly large, with many different sections and data to check. While I can test the functionality, and will build and test the app in small components, I can't properly test the UX (user experience) as I'm not a member of the target market. Instead, as the project nears completion, I'll need to provide my client regular builds with UI functionality intact. Doing this will allow me to get a better sense of what is wanted by him and his colleagues (who will also test it), and allow me to make refinements. This needs to be in an unobtrusive manner so as not to take up too much of their time. I'll therefore give them areas to focus on and give feedback for with each build. Once they start to use the app with clients, they may need to use the gym's equipment; in particular, if they are testing a particular exercise, the equipment needed will be under high demand. This will need to be addressed during the testing period.

Design Mockup

In order for me and my client to get a better sense of how the app would operate, I used Xcode's storyboards to layout the flow of the app. Storyboards were a simple way to allow me to have an app running on a target device quickly and without writing code.

A storyboard also has the advantage that it uses native user interface (UI) elements; for a user to feel comfortable using an app, Apple, the platform holder, advises in their [Human Interface Guidelines](https://developer.apple.com/library/ios/#documentation/UserExperience/Conceptual/MobileHIG/Introduction/Introduction.html) (developer.apple.com/library/ios/#documentation/UserExperience/Conceptual/MobileHIG/Introduction/Introduction.html) to follow platform conventions. When apps operate consistently on a platform, using native UI elements, the user can learn what to expect when they perform a particular action.



At the time, I simply laid out using storyboards what I'd put down in my initial ideas. The purpose was to give myself and my client a clear idea of what the target was. I did not change the default appearance (including size or colour) of any UI elements, as at that stage my focus was the flow between different parts of the application.

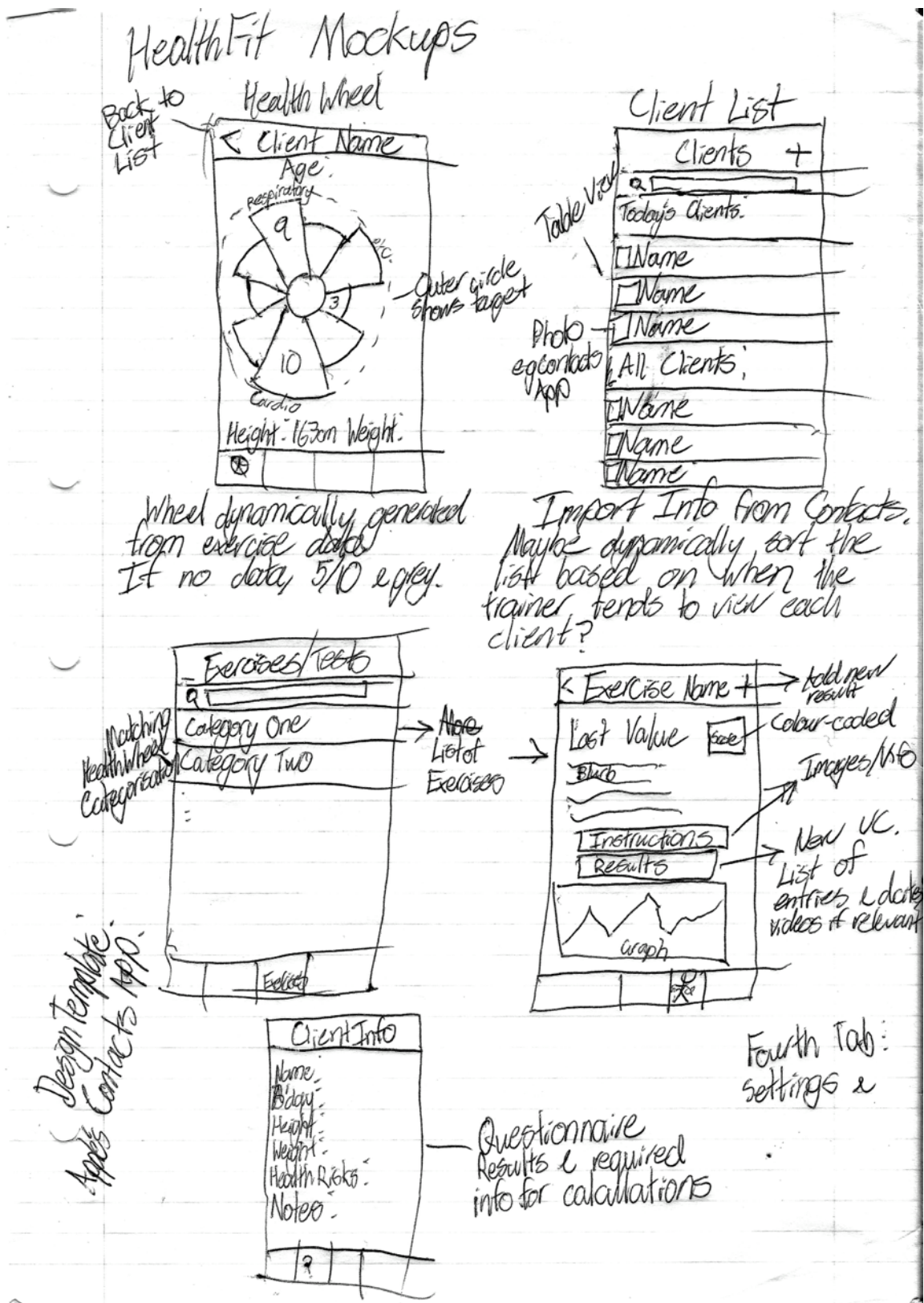
This basic mockup gave me an idea of the number of primary view controllers that would be necessary to code, and thereby some guess at the scope of the app.

I showed the basic application layout to my client on a device, and he affirmed that it met his needs and reflected his vision for the app's functionality.

I then drew some design sketches of how the different views could potentially look. Using the storyboard as a template, I drew in the rough positions of the app-specific elements. This allowed me to clarify where the different data stored in the app is displayed. It made me realise that I may need more views than I had anticipated – for example, each exercise will separately need a way to view information about the exercise, a way to view data, and a way to view past results, whereas in my storyboard I had all this information contained within a single view.

Unlike my initial idea, I realised through critical reflection that creating and then doing workouts might not be the best workflow; it would be easier to allow the user to do each assessment individually, on their own terms, rather than being limited to a particular section, having pre-selected their assessments. This removes an arbitrary restriction, making the app more versatile for the user and allowing them to develop their own workflow.





An issue particular to digital platforms is that an interface may appear different to users depending on their device. There are, at this stage, four different form factors (iPhone, iPhone 5, iPad, iPad mini) on which the app may run, and on each it needs to appear as if it were made specifically for that form factor. There are frameworks such as Auto Layout (where a developer puts constraints on the position of views relative to each other) that can mitigate this issue, but the reality is that I can't know exactly how the app

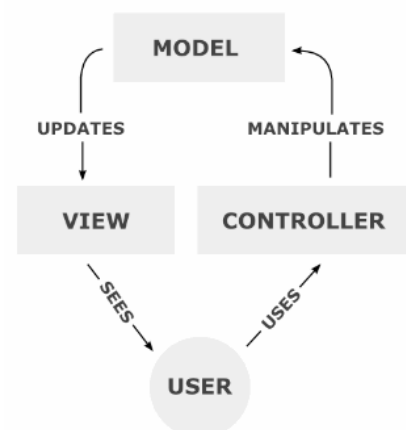
will look until the code is running on the device. Having an idea of what goes on which screen, rather than the exact positioning of elements on said screen, is therefore the focus.

Research

Core Data

The app is going to need to store a database of customer information, keep it secure, and have it readily accessible. For this, I plan on using Core Data, an Objective-C API that both wraps SQLite and extends it through methods that allow it to automatically populate a table view, for example - very useful for a list of clients. In addition, using Core Data theoretically opens the possibility for internet synchronisation through Apple's iCloud, although the reliability of this in its current state has been called into question by many experienced iOS developers. I intend to use it to provide the majority of the model portion of the model-view-controller design pattern, a pattern enforced by the iOS frameworks and encouraged for clean, tidy code. This also contributes to the client need of code that could be maintained by a different developer - due to its ubiquity, a future developer would likely be already very familiar with the MVC design pattern.

To prepare for using it in a second production app, I watched the Apple's 2013 WWDC session video on Core Data performance, optimisation and debugging. The video covered common performance-issue scenarios and resolutions for these, and will be very useful when it comes to writing my code. It also covered how Core Data could be used in a similar app to what I'll be making and gave some ideas on how I might design my code around using the Core Data framework.



Contacts Integration

One of the attributes required for the app was integration with the Contacts app. I have never attempted to use the Address Book framework, so I needed to investigate how it is used. I looked at [Apple's documentation for the framework](https://developer.apple.com/library/ios/#documentation/ContactData/Conceptual/AddressBookProgrammingGuideforiPhone/Introduction.html) (developer.apple.com/library/ios/#documentation/ContactData/Conceptual/AddressBookProgrammingGuideforiPhone/Introduction.html), and checked to make sure it could do the key things required by my app:

- Load a person's information, such as name, email address, or birthdate, into the app in formats usable by the standard APIs.
- Display a modal view controller, allowing the user to input a new contact for inclusion in the Contacts app.
- Save modifications and miscellaneous data into the Contacts database - for example, height might be stored in the same file as the base information. Some information will

need to be split between the local SQLite database and the Contacts app, so I'll need to plan how the information will be divided.

Thankfully, Apple's documentation covered all of these areas but the last; custom data will need to be stored within the app. The emphasis in the documentation was on using the Apple-provided controllers rather than accessing the data programmatically, although the latter is still supported, and any direct access is through a low-level C based API. It might be a good idea for me to consider wrapping these function calls in higher-level Objective-C methods to speed up development time, particularly if I'll be frequently calling those functions. This will require further research.

Camera Integration

Another feature that my client requested was the ability to take pictures or video and attach them to exercise data. There are two main components to this: a) exposing the native camera UI and allowing the user to use it from within the app, and b) saving the data to disk within the app's sandbox, so it's kept even if the user deletes the photo from their camera roll. This is accessed on iOS through either the [AssetsLibrary](https://developer.apple.com/library/ios/navigation/#section=Frameworks&topic=AssetsLibrary) (developer.apple.com/library/ios/navigation/#section=Frameworks&topic=AssetsLibrary) framework (to retrieve photos or videos that have already been taken), or through the UIImagePickerController and UIVideoEditorController classes provided within the UIKit framework (documentation at developer.apple.com/library/ios/#documentation/AudioVideo/Conceptual/CameraAndPhotoLib_TopicsForIOS/Articles/TakingPicturesAndMovies.html). The classes appear to be very similar to the email composition view controller, which I've used before. I therefore think that implementing this will not be a particular challenge.

My technology teacher suggested that it might be useful for photos to be annotated within the app. Although my initial response was that it would be difficult to lay out the UI for something like this, I nevertheless recognised the potential value to my client. A feature such as this would be an addition, rather than an integral part of the app, so while it may be useful, it is not a priority to have for the prototype, where having the app usable as soon as possible is a major priority for the client.

iOS 7 UI Concepts

Recently, Apple have redesigned iOS and changed the way many of its UI elements work. This new release, iOS 7, is due out this September, approximately my target for completion of this progress.

I watched Apple's WWDC session video on "Building User Interfaces for iOS 7". The main takeaway from this session was: focus on the content, and allow the UI to give way to the content. The new visual language has an emphasis on conveying information through sparse, bold colours, thin typography, and physics-based animation - the latter to such a degree that there is a new framework, UIDynamics, based around physics interactions.

The new design of iOS presents a major opportunity for new apps. Existing apps need to be updated to look current on the new OS, which in some cases constitutes a full redesign of the app. New apps can take advantage of the new design without having to

worry about backwards compatibility, and can have major differentiating features as a result – for example, they can use iOS 7 only typography features which drastically alter the look of the app. This is an opportunity for disruption; an app which seems well designed and fits on the new OS should be more attractive to its users.

A major issue for feeding back my progress to my client is that I am, at the time of writing, under a non-disclosure agreement with regards to discussing information about beta versions of iOS that prevents me from disclosing information that has not been publicly disclosed by Apple. My technology teacher is under the same agreement, so showing him my work is not an issue, but it could prevent my client from being able to test the app or give feedback. Because of this, I plan to focus on functionality first and use standard system elements in my design until the app is feature complete, so he can still see and test the app. By the time of release, more information will have been released about iOS 7, so I can then focus on the appearance and user interaction, getting regular client feedback. This also should lead to better code – if I’m focusing on having a flexible UI, then the UI code will be compartmentalised and therefore easier to maintain.

Script Evaluation

The app will have a large number of exercises stored in its internal database. Each exercise is unique, and will have a unique algorithm to convert the results to a number between one and ten. I don’t want to have to hard-code each exercise into its own class within the app because of code complexity and maintainability; not only would it be difficult to modify, but it would also increase the size of the binary and mean that the code structure of the app would easily become muddled. Ideally, I want to store the exercises in an XML or text file, with the algorithm associated with the exercise. The key-value structure of a dictionary seems ideal for this, so an XML property list might be the ideal format to store all the exercises: each exercise would be an individual dictionary, and the algorithm to convert from the results to a number would be a string.

If I am to have each algorithm as a string, the app needs to be able to evaluate that string. It can’t be in the format of Objective-C code, as Objective-C needs to be compiled. I therefore have two options: use a scripting language to evaluate it, or write them in a custom format that can be evaluated with a custom parser. The latter could be more simple if I don’t need flow control (e.g. if, switch) in the scripts, but at this stage I’m not sure what the scripts will involve. Therefore, to avoid limiting the algorithms or the exercises possible for my client to add, I think the former option, that of using a scripting language, would be best.

There are interpreters bridging Objective-C and scripting languages for almost all of the commonly used languages. I don’t have a preference for any in particular. One of my aims is to keep the code as simple as possible in case my client wants to move forward with a different developer, so I think it would be best to use the only interpreter built into the iOS libraries: a JavaScript interpreter. This also has the benefit of having [many articles written about](#) how to use it. I’ll write the algorithms in JavaScript, pass the variables necessary in to the script, and get the result out as an Objective-C `NSNumber`. This will allow easy editing of the exercises within having to modify the main codebase.

Password Security

One need that the client outlined was the ability for the app's data to be secure; personal information about the clients would be stored and be under confidentiality. To prevent unwanted access, I think that optional password protection for the contents of the app would be the best option: when the app is opened or woken from sleep, it would prompt for a password. The Core Data database would therefore need to be encrypted so the data within it couldn't be manually extracted.

In this area, the desires of the primary and secondary stakeholders may clash. My clients may want to be able to access all their clients' data at any time to prepare for sessions, but the secondary stakeholder (i.e. their client) may prefer that their personal information is not made available for the trainer to see when they are not present - in other words, that each client has their own password. In this area, the technological demands of storing multiple passwords for the database, restricting access when necessary, coalesce with the inconvenience this would cause the trainer, resulting that the only reasonable option is to have a single password for the app.

Encryption is a complicated area, and it's one where I don't trust that I have enough experience to do it well. I found an open-source framework which works to encrypt the Core Data database with minimal work for the developer: [Encrypted Core Data \(https://github.com/project-imas/encrypted-core-data\)](https://github.com/project-imas/encrypted-core-data), which internally uses [SQLCipher \(sqlcipher.net/ios-tutorial/\)](https://sqlcipher.net/ios-tutorial/). I think that using these libraries, rather than implementing the encryption myself, would be the best solution for both myself and my client.

Stakeholder Input

I asked one of the personal trainer's clients, whom I knew personally, for what they might expect from an app designed for the use of their trainer. Two main points stood out.

- Tracking progress is exceptionally important. People go to trainers to improve, and they want to see themselves improving. For this, the client indicated his preference for clear, easy to track graphs. I mentioned the idea of the progress wheel, and the client expressed enthusiasm, although he did mention that there need to be clear indicators as to what individual components might be holding them back from a higher score. When tracking multiple statistics for a single exercise (e.g. mass of weights and number of lifts) it would be fine for the graph to display the number produced by the algorithm (between 1 and 10), rather than displaying the actual values.
- Allow the personal trainer to mark particular exercises as important for a client to continue, and have the app batch them together in a single email with images and send them to the client as instructions. This would be a valuable feature in my opinion, and should be reasonably trivial to implement.

I discussed the latter feature with my client. His response was that the exercises are more assessments than exercises, and, with a few exceptions such as resting heart rate (which should be taken immediately after waking up), they should be done in the presence of the personal trainer. Instead, it would be more valuable to send the trainer's client a

summary email of their progress, rather than getting them to assess themselves through potentially unclear email instructions.



Final Brief

Conceptual Statement

I am to make an application for personal trainers to record and track the progress of their clients over time. I am making this to fill a gap in the market for apps focused on the personal trainers themselves rather than their clients.

Specifications

- The application will be built using Xcode and written in Objective-C, and, according to the client's wishes, be built to be compatible with iOS 6 and 7. The supported devices will be the iPhone 3GS and later, the iPad 2 and later, and the third generation iPod Touch or later.
- The app can store a database of clients using Core Data for performance and stability. These clients can be displayed in a table view, with a list of names and photos. The app will import any new data for a new client from the Contacts app without the user being asked if a client is entered who has the same name as a contact, to save the user time entering data (subject to the user giving access to the Contacts). It will also store certain attributes about each client. What these attributes are will be decided by my client, but can be easily altered in the Core Data model – a benefit, as the client has not completely decided all which he wishes to have an option to track. Using Core Data fulfils the client needs of having a secure backend for a responsive app, and the functionality meets his need for having multiple clients. Using a pre-built framework also reduces development time, another priority.
- The client information view (the second tab along) will contain the editable results of the questionnaire, stored using Core Data. This meets the client-specified attribute of the app digitising the client's questionnaire, which would have had to have been entered manually on paper. This increases efficiency and reduces the possibility of data loss.
- The app contains a built-in database of exercises. These will be provided by my client, and accessible through a table view within the app for each user. Each exercise can have associated results of a certain type, which can be input into the app and viewed, either individually or graphically. Every user will have different results for each exercise.
- The exercise results will be processed into a numerical value for each category. This will be displayed as a daisy wheel graph on the first tab for each client. This meets the secondary stakeholder need of being able to visually track progress.
- Each exercise needs to have an algorithm associated with it to convert the data for that exercise into a number for the health wheel. This will be stored as a string for each exercise, relating to a JavaScript function that can convert the input result into a value. This function means that the data for the exercises can be stored separately from the

app's main source code, keeping it compartmentalised and maintainable for a future developer.

- The app will have a tab where the personal trainer can enter extra information about each client in a table view, themed like the Contacts app's editing view for familiarity. This information will be retrieved from the Core Data database.
- The app needs to be secure – the database should be ideally encrypted and password protected, requiring the personal trainer to enter their password before they use the app.
- The app needs to be able to summarise relevant information about a client and their progress, including any notes from the trainer, in the standard format used by personal trainers.
- The app will need to be able to take and store images or videos of clients performing exercises, using the iOS Camera API. These images would be displayed in the individual results section for each exercise.
- The app needs to be able to be themed to suit different companies' branding; in other words, the controls for any branding images or colour schemes must be exposed within the app's settings, or else easily configured in the code, for the user's company.
- The app's UI will use native elements wherever possible and aim to take advantage of the new iOS 7 APIs in order to add value to users with updated devices.

These specifications work together to satisfy all the attributes listed by my first client in our initial meeting. The target would meet key needs and increase efficiency for the client, representing a clear improvement over current, paper-based solutions.

Overall, this application concept will meet the initial aim: allowing personal trainers to easily and effectively show their clients their holistic health and progress. The concept respects the context of the iOS digital application market, whilst simultaneously meeting the needs of the primary stakeholders. It is therefore fit for its purpose. Looking forward to the prototyping stage, I will need to make sure to continue balancing the needs of the stakeholders with the restrictions of the context.

Prototype Development



Client Meeting

Before I began to build the prototype, I met again with my client and presented the final brief to him. He approved the brief as it stood, given the understanding that the design and implementation of some of the elements of the exercise tab in particular were subject to change, and that the overall design of the app in terms of customised appearance was non-final. With that established, I began development.

Stages of Development for the Initial Screens

As I have planned the functionality of the app and the overall design, the next step is to plan how I am going to build it – the stages of development, and the layout of the code. I also need to design individual views.

I have decided to build the app in distinct, modular stages, corresponding with a body of code or a piece of functionality within the app.

1. Plan the object relationship layout – the different objects, their properties, and how they interact – in the visual editor within Xcode.
2. Implement the application back-end (the model layer) in code, including incorporating the encrypted Core Data store to meet the client need for secure information; in other words, implement step one and document the code.
3. Build the person table view:
 - 3.1. Build the functionality to add people without using Contacts.
 - 3.2. Add a search bar.
 - 3.3. Add importing from Contacts.
 - 3.4. Implement client deletion.
 - 3.5. Populate and format the table view.
4. Implement the health wheel. It needs to be animatable based on test data – it cannot use actual data until a later stage.
5. Implement the Results tab. This is a large stage that will need to be separately planned after the previous steps are implemented.



The Splash Screen

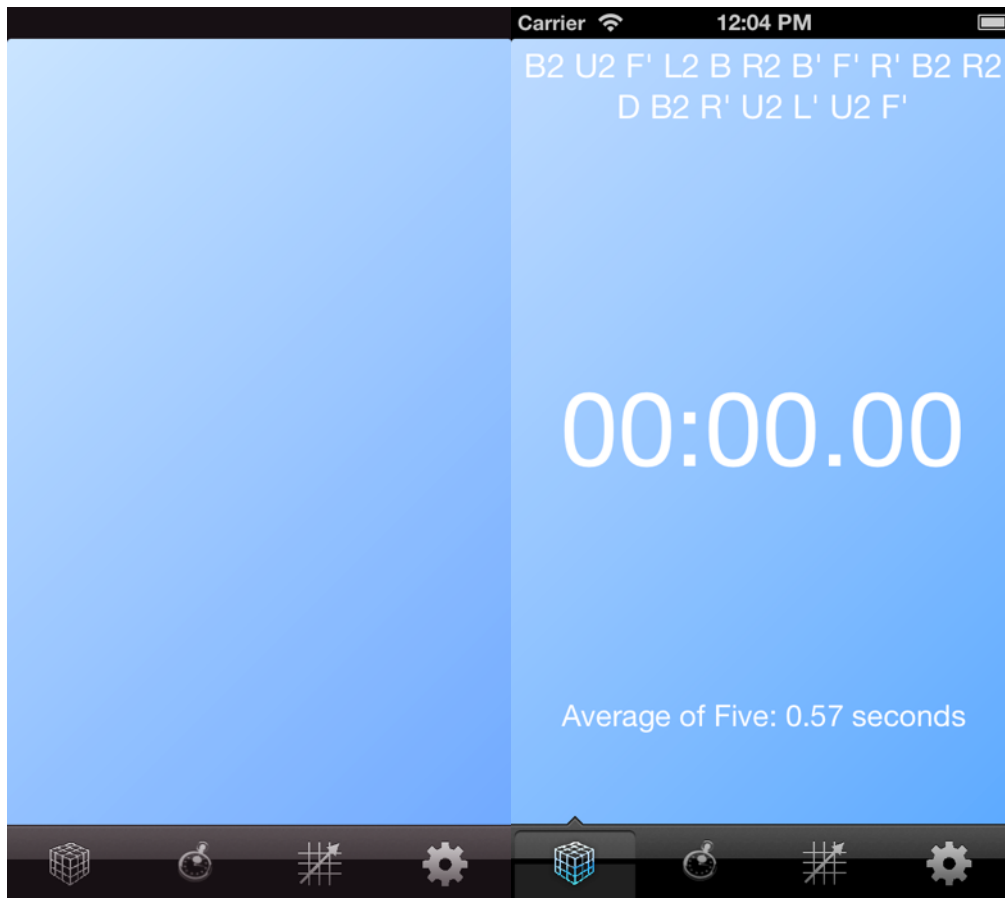
During the meeting, the client and I had discussed the concept of a splash screen – that is, that when the app is opened, it should display the logo of the company. I was opposed to this concept based on my prior experience in making apps and Apple's guidelines: in their [Human Interface Guidelines](https://developer.apple.com/library/ios/#documentation/userexperience/conceptual/mobilehig/IconsImages/IconsImages.html) (developer.apple.com/library/ios/#documentation/userexperience/conceptual/mobilehig/IconsImages/IconsImages.html), Apple say:



- A launch image looks very similar to the first screen your app displays.
- Because users are likely to switch among apps frequently, you should make every effort to cut launch time to a minimum, and you should design a launch image that downplays the experience rather than drawing attention to it.
- Generally, design a launch image that is identical to the first screen of the app.

The idea is that having an image displayed makes it seem as if the app has loaded quicker than it has in reality, and gives the expectation that the app is still responsive and not 'frozen.'

I had followed these guidelines on my previous app, CubeTimer; on the left below is the image that is displayed while it is launching, while on the right is the app's possible appearance once it has loaded.



Shortly after the meeting, I received an email from my client. The relevant portion went as followed:

Hi Thomas,

I've spoken with the other shareholders in regards to the cover image to the app. and we all agree that we'd like to please have this included even though this will have a slight delay on speed of the app.

*The reason being is that our industry relies, uses and values a lot of tools that have a look of **professionalism** and **sharpness** to them. We feel that including a cover image with our branding will not only **market our brand** BUT also create a **look and feel** for the App. itself.*

Having read this, I understood what he was trying to achieve, and in the above email, I have bolded the important points as I identified them. In regards to the splash screen, however, Apple's guidance and my experience with apps as an end user still left me in disagreement with the idea of including a splash screen as opposed to an unobtrusive launch image. As such, I replied the following:

I can produce a cover image if you like. I still stand by what I said, but it's simple enough to make two versions so you can decide which is better when you're using the app. However, what you said about professionalism, sharpness, and giving the app a look and

feel – that's not going to be created by a cover image, but instead by the interface design within the app and the responsiveness and utility of it; the mysterious element called 'polish' is what we'll be trying to achieve, because that feeling is created when an app feels polished. Likewise, branding can be shown through subtle colour cues and shapes throughout the interface. Regardless of the final decision, I am fully in agreement that we must strive for an end user experience that looks and feels professional and polished.

My client therefore agreed to postpone the final decision until such a point when the app will be ready for focus testing, basing the decision off which of the two the testers prefer.

Style Guide

In order for both legible, tidy code and for a future developer to be able to easily understand what I have written (should the client choose to continue in an update with a different developer for whatever reason), I need to follow a consistent style while coding. I have decided to adopt two main sets of guidelines:

- The [New York Times Objective-C Style Guide](https://github.com/NYTimes/objective-c-style-guide) (<https://github.com/NYTimes/objective-c-style-guide>). This will be my primary guide for coding style. It is a public resource managed by experienced developers who have shipped a widely used app, and I already follow the majority of its practices.
- Apple's [Coding Guidelines for Cocoa](https://developer.apple.com/library/mac/#documentation/Cocoa/Conceptual/CodingGuidelines/CodingGuidelines.html) (<https://developer.apple.com/library/mac/#documentation/Cocoa/Conceptual/CodingGuidelines/CodingGuidelines.html>) (Cocoa is the name of the family of Objective-C APIs used to build applications for iOS or OS X). While these are targeted at a developer designing a framework for others to use (rather than a program), they still provide excellent advice from experienced developers.

Where there is a clash between what Apple advise and what the NYT Guide advises, I will follow the NYT Guide.



Localisation

A key part of this app is the potential for expansion. While the initial usage case is for the personal trainers at my client's company, it is quite possible that the app might eventually be used across the world by different trainers. Apart from the customisation options specific for each trainer – customising the app's appearance to reflect the company – there are two main barriers to localisation. These are:

- Language. The app may end up being used in multiple languages, and for that it will need to be translated. While I cannot translate the app myself, I will need to build in the potential for [localisation](https://developer.apple.com/library/ios/#referencelibrary/GettingStarted/RoadMapiOS/chapters/InternationalizeYourApp/InternationalizeYourApp/InternationalizeYourApp.html) (<https://developer.apple.com/library/ios/#referencelibrary/GettingStarted/RoadMapiOS/chapters/InternationalizeYourApp/InternationalizeYourApp/InternationalizeYourApp.html>); this means, in code terms, using the `NSLocalizedString` macro wherever I use strings (snippets of text) in my code. In practical terms, this means that a line of code that might otherwise look like this:

```
NSString *greeting = [[NSString alloc] initWithFormat:@"Hello %@",
nameString];
```

will need to be written like this:

```
NSString *greeting = [[NSString alloc]
initWithFormat:NSStringLocalizedString(@"HELLO", @"The string displayed"),
nameString];
```

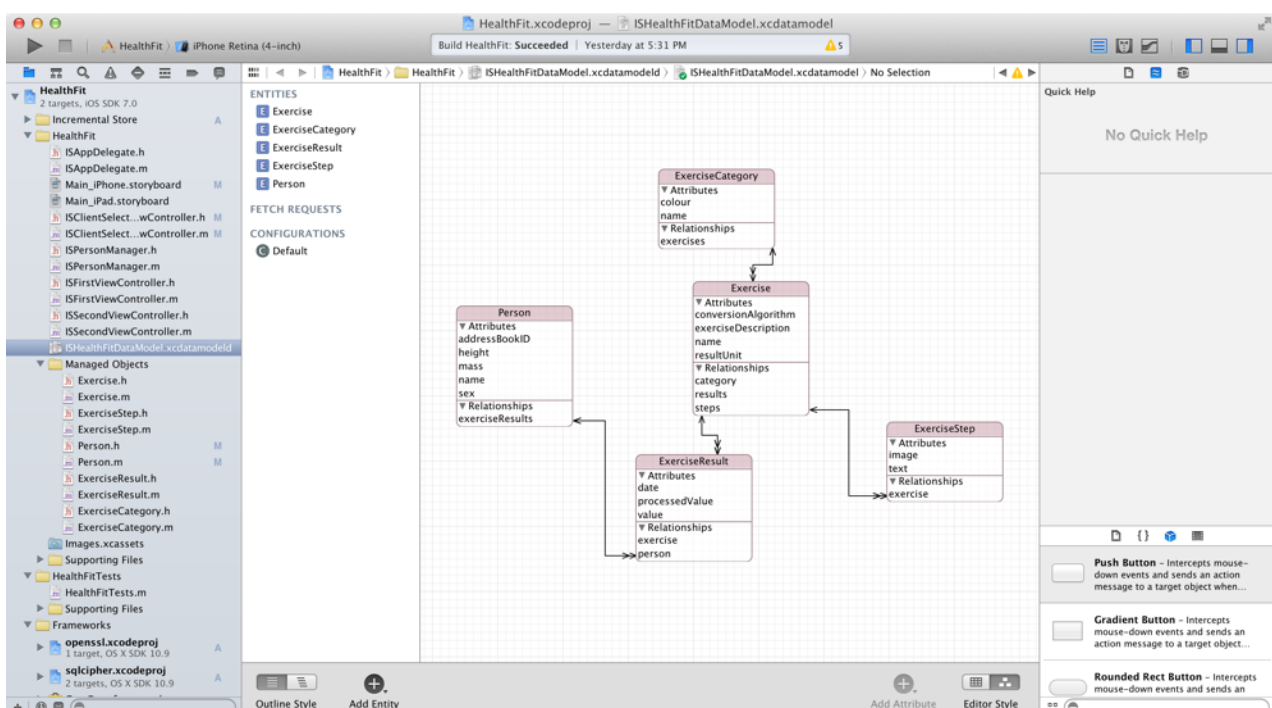
This also has the benefit of clearly showing which strings are user-facing, so they can be easily modified outside of the code.

- Unit localisation; support for both U.S. Imperial measurements and their metric equivalents. For the sake of simplicity, I have decided that internally, all measurements will be stored in SI or Metric unit where applicable (in some cases an arbitrary unit may be used; for example, beats per minute in heart-rate). As such, every time a value is entered or displayed it may need to be converted from or to the user's chosen measurement system. For simplicity's sake, I'll store all values internally in SI (standard scientific) units, so that the conversion algorithms need only expect one unit for values; however, I still need to create UI for entering in systems where a decimal point does not suffice. Feet and inches are two separate units, and somehow I'll need to find a way to convert input in that form into metres.

This localisation must occur in order to meet the client's need for expandability.

Database Planning

I began building by planning the base database structure for my app using the Core Data visual editor built in to Xcode. Deciding how this part of the app (the 'model' layer) will work is fundamental to the resulting code; having a solid basis to build around can mean that the amount of code is lessened and readability is greatly improved. The editor generates the necessary code for me to use this plan within the app; any changes that I make are 'live,' which allows me freedom in early prototyping.



Implementing Encryption

Following my research in the brief development stage, I implemented [Encrypted Core Data](https://github.com/project-imas/encrypted-core-data) (<https://github.com/project-imas/encrypted-core-data>) into the application code base, following their guide for setting it up in an Xcode project. However, it quickly became apparent that the project was not designed for password protection, and was rather supposed to use a single 'master' key to encrypt the database within each copy of the app.

On reflection, I realised that this could potentially be a better design. Having passwords introduces a range of issues, the primary one being that, should a user forget their password, there would be no way to retrieve the password and the data would be lost. Furthermore, I would have to implement extra UI to handle all the potential situations with passwords. As such, at this stage I am choosing to simply have encryption, which would still keep the client data secure, and which would mean preventing viewing is the choice of the device's owner; they can add a device passcode if they so choose. I may change this based on client feedback later, as doing so would only require me to add code and not drastically change my initial solution. This approach also means that the app will be fully functional more quickly.

When I asked my stakeholders, including my client and a potential user (i.e. one of their clients), they thought that having a per-app password could potentially be a 'nuisance' given that many people already have a passcode on their phone; needing to enter two passwords to use the app increases user friction, which in turn decreases retention.

The Client Selection Screen

Contacts Integration

I knew from the brief development stage that I would have Contacts integration within the client list. My client has estimated that perhaps half of all trainers keep their clients in their Contacts; for the others, having the app automatically add to the Contacts might perhaps cause frustration.

Originally I had decided to have a Core Data attribute for each possible aspect of a person entity. However, this would duplicate a lot of information that is already stored in an `ABPerson` instance. It could also mean that the information in the app could become out of sync with the information in Contacts.

For the initial release, I have decided to keep the Person entity as small and possible, storing any relevant information in the associated `ABPerson`.

There are both benefits and issues with using this associative approach.

Benefits:

- Attributes such as birthday, address, and contact information are all stored within pre-existing Address Book properties.
- The Address Book API supports contact images and manages storage and generating thumbnails.

- Any changes made in Address Book are reflected in-app.
- Using it keeps the Person entity simple and the database small.
- Pre-existing clients can be easily imported.
- Clients can be stored in a custom Group within Contacts, and within the Contacts app you can choose to hide specific groups.
- I can use the existing, Apple-made View Controllers to add or edit clients, which saves development time.

However, there are also many issues with this approach:

- If a person has not granted the app access to their Contacts, the app needs to be able to store the information regardless.
- If a record is deleted from Contacts, then the app needs to be able to handle that deletion without losing information, or else forbid the deletion.
- If a person does not want to have their clients in their Contacts, the app still needs to be able to function.

In solution to these issues, Apple does state that “Person records don’t necessarily have to be stored in the Address Book database.” ([ABPerson Reference](https://developer.apple.com/library/ios/#documentation/AddressBook/Reference/ABPersonRef_iPhoneOS/Reference/reference.html) at developer.apple.com/library/ios/#documentation/AddressBook/Reference/ABPersonRef_iPhoneOS/Reference/reference.html). As such, it may be necessary to keep a copy of the record within the local database in case it cannot be found in Contacts; I therefore save a copy of the contact locally in my database as a vCard format data package.

In addition, the benefits in terms of reduced development time outweigh the potential issues, as implementing a new method of storage would likely not take a large amount of time should it be done later on. In later versions, I may move the address-book based `Person` class to a subclass, having a second subclass representing a non address-book linked `Person` entity. This would allow all user-space code to simply reference them as instances of the `Person` class, although their implementations would differ; this type of encapsulation represents good technological practice.

Writing Convenience Methods for Getting Person Info

Using Address Book as a backing store for my person info does mean that accessing person information programmatically becomes more complex. For example, finding out someone’s birthdate requires the following code:

```
ABRecordRef recordRef = self.addressBookRecord;
NSDate *birthdate = CFBridgingRelease(ABRecordCopyValue(recordRef,
    kABPersonBirthdayProperty));
```

This is more code than is ideal for a task that is going to be fairly common. As a result, I decided to wrap these calls in convenience methods, so that finding out someone’s birthdate requires no more code than:

```
NSDate *birthdate = person.birthdate;
```

which is far easier to read and tidier. I have done this for any piece of information that I need to access multiple times within the Person class. The header file, containing all the declarations of methods and properties, now reads thus:

```
static NSString * const PersonEntity = @"Person";

@interface Person : NSObject

/** The person's height, stored in metres. */
@property (nonatomic) float height;

/** The person's mass, stored in kilograms. */
@property (nonatomic) float mass;

/** The Address Book RecordID for this person. */
@property (nonatomic) ABRecordID addressBookID;

/** When a Person entity is removed from memory, it stores its ABRecord
as a vCard file. This is in case the original ABRecord cannot be found.
*/
@property (nonatomic, retain) NSData *vCard;

/** The Address Book RecordRef for this person */
@property (nonatomic, readonly) ABRecordRef addressBookRecord;

/** This person's sex. Represented internally as a boolean. Used to
calculate category scores. */
@property (nonatomic) Sex sex;

/** The person's full name. */
@property (nonatomic, retain) NSString * name;

/** All exercise results associated with this person in an unordered
set. */
@property (nonatomic, retain) NSSet *exerciseResults;

/** Returns the person's age. */
- (NSInteger) age;

/** Returns the person's birthdate. */
- (NSDate*) birthdate;
/** Sets the person's birthdate. */
- (void) setBirthdate:(NSDate*)birthdate;

/** This person's thumbnail image. Can be nil. */
- (UIImage*) thumbnailImage;


@end
```



```
{
    return tableView == self.tableView ?
    self.fetchedResultsController : self.searchFetchedResultsController;
}
```


I implemented his full solution in my code with minor modifications to suit my context, and as a result the search was functional. One downside to his solution is that it rebuilds the search `NSFetchedResultsController` for each search query; if this becomes a performance issue during testing, then I may need to alter the method to modify the search query instead.

Proper Encapsulation, or, “So this is how model objects are supposed to be used”

In my previous app, CubeTimer, I’d gone through multiple stages of development. Initially, I was simply building functionality from the ground up, piling new features onto view controllers and inserting `retain` and `release` statements for memory management wherever the profiler said there was a memory issue. Predictably, this did not turn out well; the app crashed, and the code was difficult to read and hard to maintain. So, when the time came to fix these issues, I went to a developer who used to work at Apple for advice. He told me to scrap the codebase and rewrite it using proper encapsulation, so I did. 

The end result was much nicer, and it’s the code I’m still using now to maintain the app. However, the codebase still grew immensely; there was now more encapsulation, but also more complexity, and the roles of each object were not clearly defined. I used a singleton context for my database, and this object belonged to the App Delegate, seeming as I had nowhere else to put it. Accessing it, a very common task, was so complex that I had to use a preprocessor macro; `CTSESSIONMANAGER` became shorthand for `((CTSessionManager*)[(id <CTSessionManagerOwner>) [UIApplication sharedApplication].delegate sessionManager])`. Needless to say, this was less than ideal.

Building on my learnings from CubeTimer and the calculator program I built for DTS earlier this year, I decided to go back to the drawing board for the layout of this app. One thing stuck in my head; previously, I had been very careful with object allocations, thinking that they had a noticeable performance impact. But then I, watching a WWDC session, heard the speaker talk about the negligible cost of instantiating large amounts of objects, and doing it freely for code maintainability. This philosophy stuck in my head while I was creating this app.

The second realisation was that I could give my model objects, which I had previously treated as inert, struct-style data types, methods; I didn’t actually need a singleton manager. Why should I have to ask a manager what the age of a person object was when I could simply ask the person object directly? And what code does need to be in a view controller? Should I, as I did in CubeTimer, configure a data source for a graph within a view controller? Shouldn’t that be the data source’s responsibility? 

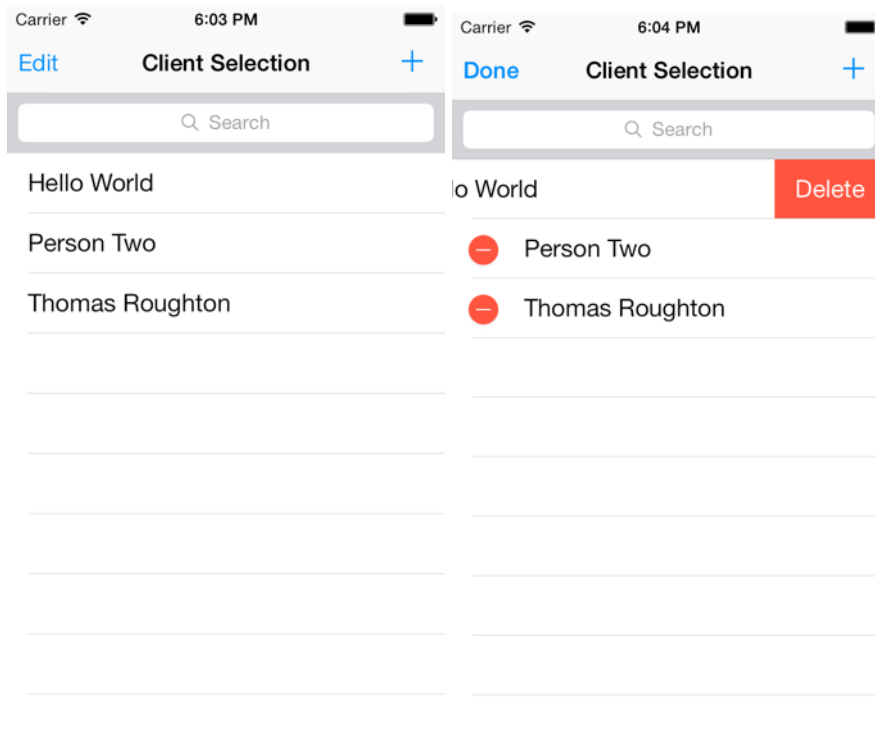
And so I applied that principle to the app. The person object should handle its contacts integration; that’s its role. An exercise knows what its `NSManagedObjectContext` is, so it, rather than the singleton, should fetch its most recent result. View controllers shouldn’t

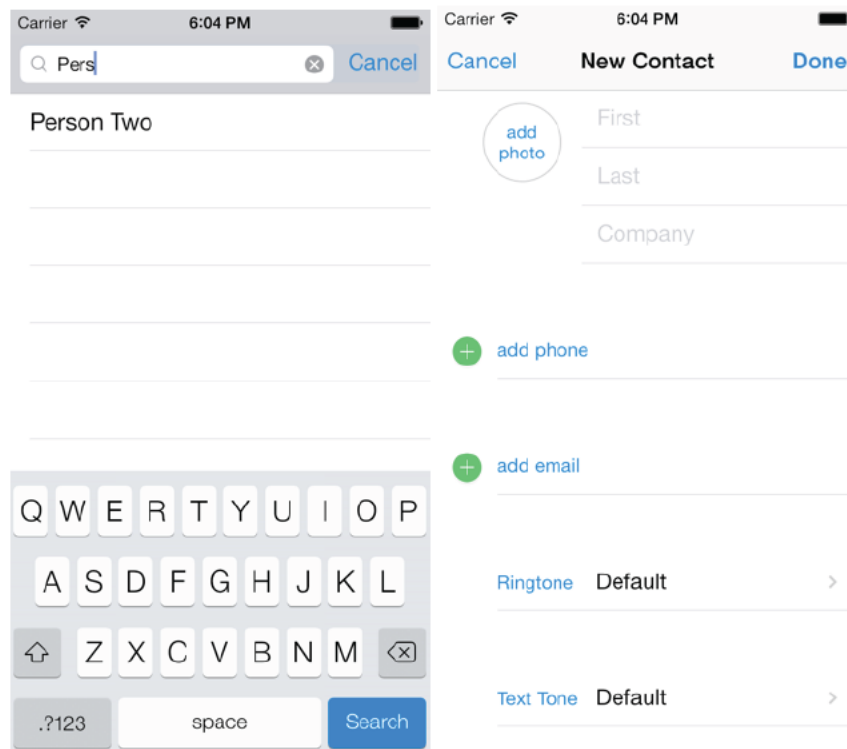
need to access the database directly; instead, I now deal directly with objects, and allow the framework to manage the database access behind the scenes. This all results in a drastic simplification of the code, making it cleaner, easier to modify and maintain, and quicker to develop. Allowing objects to only deal with their own responsibilities also makes the structure of the app much easier to visualise, which is very important; only a small part of code is actually writing the code.

The Functional Client Selection View

Once I had implemented all these things, the client selection view was functional for testing, although not necessarily complete as the visual design is non-final.

For each feature I added, I made sure to test that it worked both on my iPhone 4 (running iOS 7) and in the iOS simulator. I added myself as a contact from a pre-existing record in the devices' address books, and I also created other people from within the app. I also checked to make sure that deletion was working and that any changes made were permanently saved to the database.





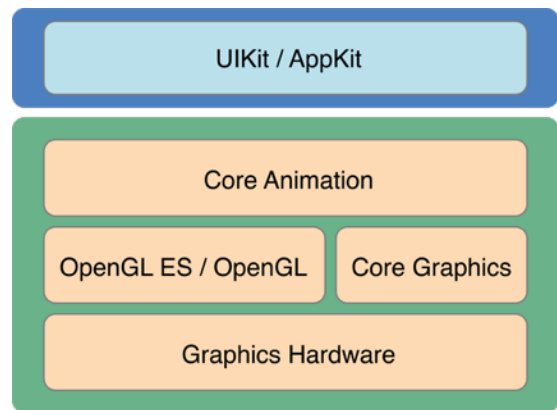
The Health Wheel

As identified in the brief, visual representation of the client's progress is very important for both the trainer and their client. In the initial meeting, my client had described something very similar to a standard pie chart; the main difference being that, instead of the arc lengths reflecting a percentage of a whole, each sector instead has a radius proportional to the client's ability in that sector. This type of pie chart is called a '[polar area diagram](https://en.wikipedia.org/wiki/Pie_chart#Polar_area_diagram)'. (en.wikipedia.org/wiki/Pie_chart#Polar_area_diagram)'



On the left is a standard pie chart, and on the right is my target mockup for how a 'Health Wheel' might look.

Animation is a key component of dynamic and interesting interfaces, so having it be easily animatable – for example, animating the growth in particular sectors between two different times – is a priority for the health wheel. iOS uses the Core Animation framework as the underlying API for all UI elements, and so I thought I'd utilise that for the backing of my views in the Health Wheel chart. This means that any changes to the charts data will be automatically animated, which opens up possibilities for additional features in the realm of comparing results.



In order to get the basic chart working quickly, I found a tutorial for a [Core Animation based Pie Chart](http://blog.pixelingene.com/2012/02/animating-pie-slices-using-a-custom-calayer/) (blog.pixelingene.com/2012/02/animating-pie-slices-using-a-custom-calayer/), whose instructions I followed. I then set about adapting it to my needs.

The first step of this was setting the angles to be the same for all slices of the chart. Next, I had to make the size of the slices vary depending on data passed in. I did this by applying a single, animatable `CATransform` to each slice:

```
slice.transform = CATransform3DScale(CATransform3DIdentity,
    sliceRadius, sliceRadius, 1);
```

where `sliceRadius` is a value between 0 and 1 of how large the particular slice is.

I knew that I needed a way to easily set data on the wheel view. The `WheelView` class from the tutorial took an array of `NSNumber` instances and set up the pie chart based off that. For my purposes, I need more than simply a size, however; I also need to be able to tell the chart what colour each slice should be and the title for each slice. I therefore created an `ISWheelValue` class to hold these values, and passed an array of instances of that class to the wheel chart.

Once this was implemented, I realised that there was an issue with this particular method of presenting data; when the scale is completely relative, there is no way to tell what the target is. To remedy this, I simply drew a dashed circle around the outside of the chart, showing what the target is.

I tested the view by populating it with random data and having that data change every few seconds. I showed this demo to my client as an example of the wheel's operation.

At this point, the only remaining thing to implement for the chart was to add labels. There were a few different possibilities for how this might look.

- I could have labels around the outside corresponding to each slice. The text would be curved and rotated to match the arcs of the dashed circle.
- Instead of the labels being around the outside of the dashed circle, they could sit around the outside of the slice.
- The labels could be just within each slice as contrasting or white text.
- The labels could be horizontal, sitting just to the side of each slice.



All but one of these options had major flaws when I tried them. With horizontal text, the text ended up being cut off on the side of the screen. For the second option, the text could end up overlaying the dotted line, which would not demonstrate the desired 'polish' and elegance. The most major problem, however, was that in the case of the second and third option, the text would be too small. A simple mathematical formula shows mathematically what makes intuitive sense:

$$s = r\theta$$

This formula relates arc length (s) for any circle to its radius (r) and the interior angle (θ). For the Health Wheel, the angle will be determined by the number of slices;

$$\theta = \frac{2\pi}{n}$$

where n is the number of slices. The number of slices will be constant, so the arc length, which is directly related to the size and spacing of the text, is proportional to the radius – in other words, in options two and three, the smaller the radius is the smaller and less legible the text will be. As such, the only viable option is the first one.

However, when I went to implement this, I realised that having curved text would not be as easy as it had seemed. The `UILabel` class, which is what is primarily used on iOS for non-editable, single-line text display, does not have any support for curved text. I thought that wanting to do this would not be uncommon, however, so I searched for a solution. Once again, a user on StackOverflow had [posted a solution](https://stackoverflow.com/questions/3841642/curve-text-on-existing-circle/7114184#7114184) (stackoverflow.com/questions/3841642/curve-text-on-existing-circle/7114184#7114184). They had adapted Apple's OS X sample code for curving text using the CoreText framework to the iPhone, and had packaged it neatly into a reusable `UIView` subclass.

When I implemented this, there were positioning issues; I could not get the text to correctly arc around the outside of the dashed circle. I had to fairly extensively modify the drawing code, but eventually I got it working. The end result was an almost perfect match for the mock-up I had made in Photoshop.

I created a data source protocol for the view, and randomly generated a number of random values to be displayed on the wheel. I then had the values change every four seconds to test the animation. Based on that, I realised that, due the centre of the wheel being a filled white circle, any low values (below around two out of ten) were being cut

off. I amended the code to account for the middle circle. The animations worked smoothly in every case, giving the interface a sense of dynamism.

By this point, the overview screen was almost complete. The remaining changes – populating the chart based on the results of different exercises, displaying any specific client information (such as warning flags if the client needs medical attention before further training), and any visual customisation – would all have to wait until the exercise functionality in the third tab had been completed.



Going Back to the Client List

There was a small issue with this: once you'd selected a client, how were you supposed to go back to the list? The tab bar view controller was presented modally; that is, it slides up from the bottom of the screen. Logically, you'd think dismissing it would be a matter of sliding it back down, and this happens to be a solution that works quite well. There are no major touch targets on the overview screen, apart from the tab bar, so the user is free to swipe.

I initially tried to solve this using a simple swipe gesture recogniser; when the user swipes down on the screen, the view would dismiss back to the contacts list. This worked fine on the simulator, but when I tested it on my device I found a real issue in terms of the user experience. The app was only responding after the user had performed an action, rather than while the user was performing it; rather than directly manipulating the interface, as is natural on a touch screen, it felt as if you were simply giving commands. So I tried a different solution, using a new API from iOS 7 and a tutorial I found [here](http://www.teehanlax.com/blog/custom-uiviewcontroller-transitions/) (www.teehanlax.com/blog/custom-uiviewcontroller-transitions/). I had to learn how to use the new view controller transitioning APIs, and, by both reading that

tutorial and watching the WWDC session on view controller sessions, I was able to implement the functionality into my app.

One difficulty I found was that, in following with the tutorial, the position of the screen would 'jump' to where the touch was tracking, as opposed to being dragged down with it. I modified the code so that it computed the offset from the original touch, and animated the offset of the view so that it was equal to the touch.

The second issue was visibility of the gesture. Every gesture needs to be taught to the user, and one way to do that is through subtle UI hints. iOS 7 heavily uses blurred layers to provide context to the foremost view, and I thought I would prototype that using a basic transparency effect on the foremost view (the overview screen). As the screen slides into place, the white background would slowly become opaque, showing the user that there is still a view underneath it. This provides valuable context to the user's actions, and ends up being quite a subtle effect. When I tested it on my iPhone 4, performance suffered as a result; to simply render the blur image took 420ms, where I find anything over about 100ms tends to be a noticeable delay. I'll therefore need to disable the effect on slower devices; given I'm using Apple's code to render the blur effect, I anticipate that further optimising it would not achieve significant performance gains.



This method of visual indication demonstrates originality in design and consideration of the user needs.

The Questionnaire

When a new client is created within the app, they have to complete a screening form. This acts as both a data source for the results calculations and a prompt for the trainer if there are issues within the entered data.

Having talked with my client, I made a list of the things that this screen would have to do:

- Retrieve important pieces of data from the related **ABPerson** in Contacts such as name and birthdate.
- Allow these pieces of data to be edited within the GUI of the app (within the questionnaire. The GUI is to be based upon the Contacts app.
- Having custom fields such as height, notes, and medical assessments.
- Reflecting data from assessments such as a mass test.
- Clearly representing within the GUI what data is editable and what is not.

- Having certain pieces of data be permanently locked when the questionnaire is complete – the client requires this so that his clients, the secondary stakeholders, can be sure that their data is the same as when they entered it; in other words, most fields in the questionnaire can only be set once.
- Having the ability for the app to flag certain fields as matters of concern.

Designing the back-end for this table is a challenge; it would be all too easy for the code to devolve into a mess of conditionals and switch statements. I began to build an `enum` detailing all the fields for the questionnaire, like below:

```
enum QuestionnaireValue {
    QuestionnaireValueName,
    QuestionnaireValueAge,
    QuestionnaireValueHeight,
    QuestionnaireValueMass,
    QuestionnaireValueOccupation,
    QuestionnaireValueHomePhone,
    QuestionnaireValueWorkPhone,
    QuestionnaireValueMobilePhone,
    QuestionnaireValueAddress
};
```

It was very quickly apparent how large and messy this would become, reducing code maintainability for a future developer and increasing my development time. Rather than continuing to attempt to build this from the ground up, I found a library called [QuickDialog](http://escoz.com/open-source/quickdialog) (escoz.com/open-source/quickdialog) that allows for form creation based on human-readable, easily editable JSON.

The library has an in-built feature which allows it to bind form elements to a particular object using key-value coding. In effect, this means that if an object has a property such as `name`, the element can retrieve the value of that property through calling `[object valueForKey:@"name"]`. By linking a `Person` instance to the form, the form can query the `Person` for the value to display for different elements.

This dramatically simplifies the code. I modified the library so that, instead of just retrieving values from the `Person` instance, it can also set them, using `[object setValue:@"Person's name" forKey:@"name"]` – the key is the only thing that has to be manually set in the JSON. All that is required is the following:

```
{"type":"QLabelElement", "title":"Name", "bind":"value:name"} in the JSON,
self.root = [[QRootElement alloc] initWithJSONFile:@"Questionnaire"
andData:self.person]; in the view controller, and the property
@property NSString *name in the Person class' header file.
```

One issue with this approach is that it requires that every field/element in the form has a matching property in the `Person` entity within Core Data; given that the data will only be retrieved in one place, when the entity is already in memory, I thought that adding a property for every field would both slow the database and make the code less tidy and legible.

In search of a more elegant solution, I realised I could use the key-value coding system to my advantage. Instead of creating a new property or field to back every element in

the questionnaire, or instead of adding a new questionnaire entity, I could simply add an `NSDictionary` to my `Person` entity as a backing for properties that the questionnaire requires but do not exist in the person. The `NSDictionary` class handles key-value storage in the dictionary data type for the Objective-C APIs, and has two key advantages that led to me using it:

- Instead of throwing a program-crashing exception when it is asked for a value it does not have stored, it will simply return nil; a perfectly valid default value for the fields in a form.
- Any value can be set for any arbitrary key – the keys would not need to be predefined, and therefore could exist solely in the JSON.
- In the database, it could be stored in external storage, preventing it from enlarging the database and slowing operations.

I therefore added an `NSDictionary` property to my `Person` class. I then had to implement methods to pass calls for values missing in my `Person` class to the `NSDictionary`. I overrode the method `valueForKey:` (called when no property is declared with a key matching `key`) to revert to the dictionary when a value cannot be found on the person object:

```
/** If a specific key isn't found on this object, it's passed on to a
'catch-all' NSDictionary, which holds the results for the questionnaire.
These keys are generally data that will not need to be queried when the
object is not already in cache – values used for sorting should not be
stored here. */
```

```
- (void) setValue:(id)value forKey:(NSString *)key {
    if (value && key.length) {
        [self.questionnaireDict setValue:value forKey:key];
    }
}
```

```
- (id) valueForKey:(NSString *)key {
    if (key.length) {
        return [self.questionnaireDict objectForKey:key];
    }
    return nil;
}
```

This elegantly addresses the issue and allows later expandability of the questionnaire without requiring a Core Data migration.

I entered a few of the screening form's values into the JSON so that I could see how it looked. I also added the functionality to 'lock' the form – a feature my client had requested – so that the user of the app can reassure their client that their data is secure and has not been modified since they approved it.

Refactoring

When the time came to create the assessment lists, I quickly realised that I would be repeating much of the code I had used in the client selection screen's view controller. Duplicating code makes the app less easily maintainable. As such, I decided to refactor the code, pulling out the shared functionality into a new class that would act as a superclass for all my table view controllers. This would allow me to easily fetch objects from Core Data, customise the appearance of the table view cells, and filter the table view with search without needing to duplicate code.

Doing this allows faster development and cleaner code, which advantages the client in being able to add new features and prototype functionality. It is also vital for ongoing development of the product; keeping a codebase clean and well-maintained through regular refactoring improves the quality of the final product through reducing the probability of bugs.

Importing Exercises

The app comes pre-populated with a variety of different exercises, and each falls under a different category. Rather than hard-code these exercises into the app, I plan to use JSON to store, in a file for each category, all exercises in a human-readable format. Then, before the app is released, I will pre-populate the database with the exercises and bundle it with the app.

This makes sure that the client's exercise data is secure and not extractable from within the app (helped by the encrypted database); it would not be good for the client's business if another person could use their exercises and assessments in their own product by extracting data from the app.

The planned format is:

- A .json file, named the same as its category. The root element is a dictionary.
 - The *name* key and its value.
 - The *colour* key and its hexadecimal value as a string.

Carrier 3:33 PM

Screening Form Lock

Client Details

Name Thomas Roughton

Sex >

Age 17

Birthdate Nov 29, 1995

Height >

Mass 0

Lifestyle

Activity Level Heavy L... >

Weekly Hours of Work 30-40 >

Smoker ☐

Overview Screening Form Assessments

- An array of exercise dictionaries to match the Exercise Core Data entity, each containing:
 - A *name* key and its value.
 - An *description* key and its value – an overview of the exercise and its purpose.
 - A *resultUnit* key and its value as a string – for example, this might be “Mass” or “Length” for localised units, or else might be “BPM” for heart-rate, for example.
 - A *conversionAlgorithm* key and its value as a string containing evaluable JavaScript code. These will be executed in a JavaScript environment to which I will expose the following variables:
 - Sex
 - Age
 - Height
 - Mass
 - Value (the value which it is converting to between one and ten)
 - A *steps* key and its value as an array of dictionaries. Each dictionary has:
 - A *text* key and a string value.
 - A *image* key, with its value the path of the image as a string.

Scripting for Result Conversion

In the brief development stage, I had identified using the built-in JavaScript interpreter to convert the result values into a value between one to ten. However, upon trying to implement this, I found a range of difficulties which may suggest that using something else is a better solution. The key issue was passing values to the interpreter; I would have to package the values as JSON objects and include those in the script. Before I continued to develop a JavaScript-based solution, which could be a lengthy process, I thought to investigate some other options.

Of scripting languages, the two which I have had the most experience with are Lua and JavaScript, although I am not particularly experienced with either. I knew that Lua could bind to C programs, so I thought I would try to add it to my Xcode project.

Thankfully, Lua proved fairly simple to implement. I added the source files, initialised the context in a new `ISLuaEvaluator` class, and relatively quickly had the ability to evaluate strings. In doing so, I learned more about how Lua operates; it uses a stack-based, last-in first-out model, where you push a variable onto the stack and then assign it a variable name. In the C API, each variable assignment takes two statements:

```
lua_pushnumber(_luaState, person.mass);
lua_setglobal(_luaState, "personMass");
```

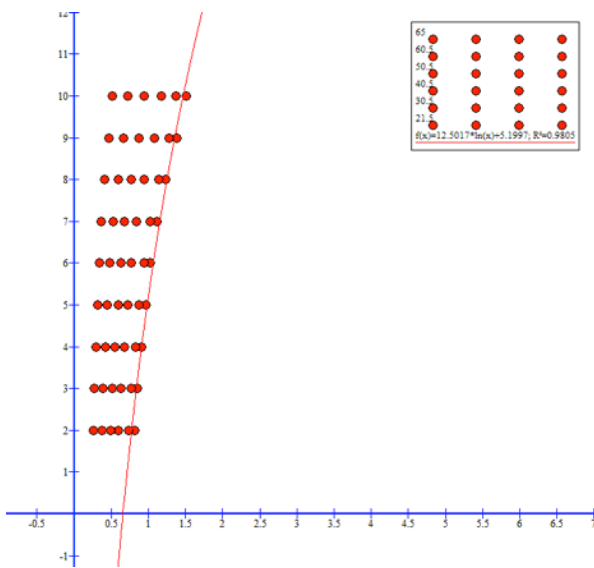
whereas, within the Lua, the equivalent statement is more similar to regular assignment, being:

```
personMass = person.mass
```

I also had to learn Lua syntax in more depth; in particular, how it handles line endings and scope, seeming as the JSON in which the algorithms are stored does not support multi-line text – marking new lines with a semicolon allowed me to work around the limitation of the format.

However, implementing the Lua interpreter was only part of the challenge. Each exercise has its own conversion algorithm, which I have to create based on pre-defined data.

	Resting HR For Men						
Age	18-25	26-35	36-45	46-55	56-65	65+	Score /10
Elite	42<	42<	43<	43<	44<	43<	10
Sub-Elite	43-48	43-48	44-49	44-49	45-50	44-49	9
Athletic	49-55	49-54	50-56	50-57	51-56	50-55	8
Excellent	56-61	55-61	57-62	58-63	57-61	56-61	7
Good	62-65	62-65	63-66	64-67	62-67	62-65	6
Above Average	66-69	66-70	67-70	68-71	68-71	66-69	5
Average	70-73	71-74	71-75	72-76	72-75	70-73	4
Below Average	74-81	75-81	76-82	77-83	76-81	74-79	3
Poor	82+	82+	83+	84+	82+	80+	2



This table needs to become an algorithm; ideally, to reduce development time, that algorithm should be fairly simple and short. Unfortunately, a table such as this does not lend itself to a single line of code if it is to be accurate.

I tried to interpolate a function between the points. Pictured left is the relationship between score and age/heart-rate. Each age group appeared as a slightly curved vertical strip, which I then tried to interpolate using a logarithmic function. I realised later that the graph in fact was an upside down graph of the form $y = 1/x$; 10 - the score is therefore

directly proportional to the heart-rate. The age only caused a slight translation. With this information, I was able to estimate an approximate algorithm for heart-rate. I may later refine this to better meet the needs of my client, but for the moment enabling him to test it sooner rather than later is of great benefit.

Accuracy of Algorithms

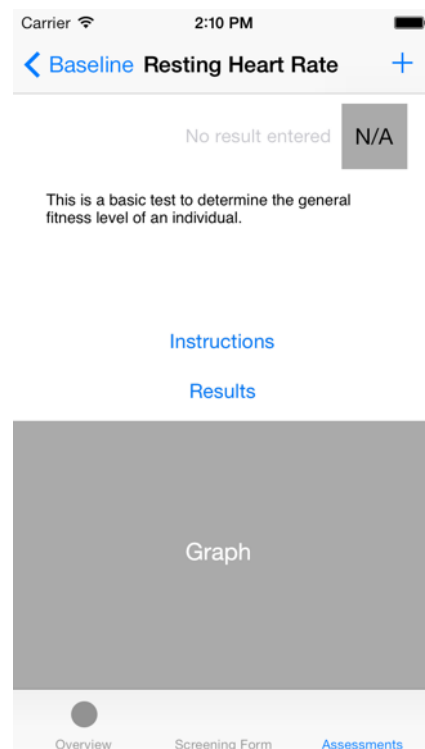
An advantage of my Lua solution is that a large number of maths functions from the built-in libraries can be used. This allows more complex algorithms that more accurately

model the desired results, and which can be refined as the app is tested. Accuracy is important, as the algorithm's result is shown throughout the result, and the trainer will likely use it to determine which areas to focus on. There is therefore an ethical responsibility to make sure that the person's health is accurately represented. An imprecise value could cause the trainer to neglect more important aspects in a person's health; if they saw that a value was low, they might focus on it, when the algorithm is in fact giving an incorrect indicator.

As the app nears completion, this will be a crucial area to focus on and refine so that the app can better reflect the client needs.

The Exercise Screen

The user needs to be able to view exercises and enter results for them. In the brief development stage, I had drawn a mockup for the Exercise view, so my initial step was to implement that interface in Interface Builder, connecting the correct **IBOutlet**s to the code so that it could dynamically adjust its appearance.



I also implemented a field so that new results can be entered and their value stored.

I contacted my client for feedback, and he indicated it might be necessary to add extra sections, dependent upon the test; for example, some tests may also need a 'Considerations' button, in addition to the 'Instructions' and 'Results' buttons.

User Considerations

I have implemented a text field on the exercise summary screen for easy entry of new results, so that the user does not have to go into the 'Results' sub-menu to view details. One issue with this, however, is that any modification to the text field leads to the entry of a new result. If a user were to make a mistake in entering the data, they would be

unable to fix it without going to the 'Results' screen and editing the object, which would potentially involve a modal dialogue. This is too much work for a common user action; therefore, I needed to come up with a solution that would allow a compromise between the ease of entering new data and the time needed to correct mistakes.

My solution was simple: have a timer between user interactions. For an app such as this, it is highly likely that a single person would not do a series of the same exercise in quick succession. As such, I made it so that if a user were to tap to edit the last result text field after only a short delay (I'm using one minute initially), instead of creating a new result it would modify the old one.

User Customisation

A key aspect of the initial brief was that the application have the potential to be visually customised by different users, so that a trainer could make the app have their branding should my client choose to license the final product.

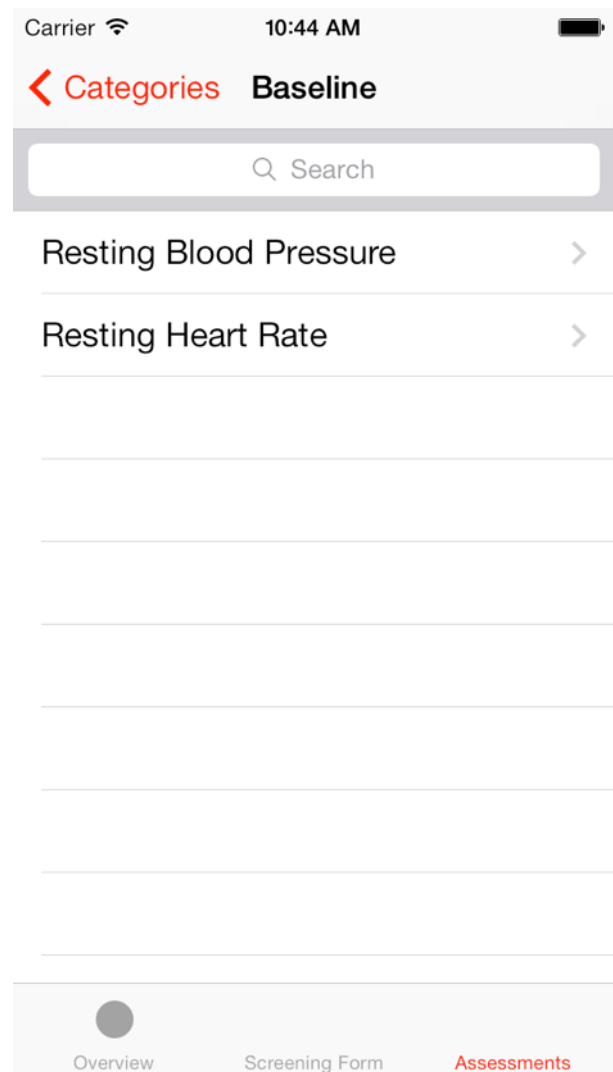
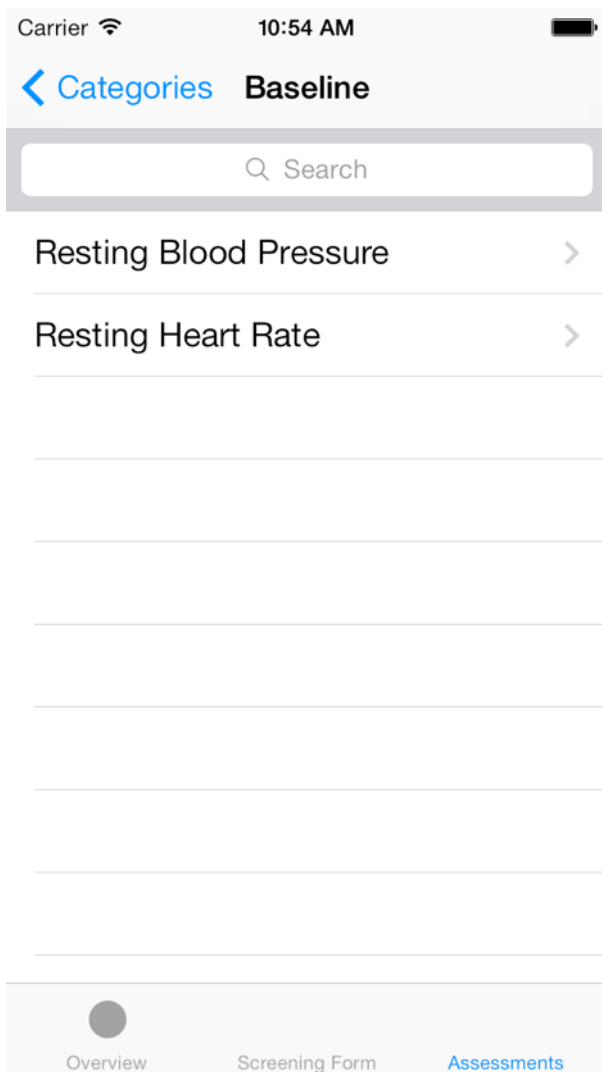
Up until this point, I have endeavoured to use standard UI elements, only customising basic attributes and approximating a layout using the drag-and-drop Interface Builder. The benefit of that approach is that implementing a custom UI element requires only subclassing an existing element; replacing the name of the superclass with the subclass in Interface Builder will cause the custom appearance to be shown, complete with working, inherited behaviour.

At this stage, however, the majority of the standard elements work in delivering the clean look that I identified to target in the brief development stage. Therefore, all that is required is to customise things such as font, animations, colour, and positioning. Rather than hard-code the desired appearance into the app, I decided to use a solution that I had read about from another developer. *Q Branch*, the developers of the iOS app [Vesper](http://vesperapp.co) (vesperapp.co), had developed a framework that they called *DB5* (<https://github.com/quartermaster/DB5>); the purpose of their framework was so that "Our designers could easily make changes without having to dive into the code or ask engineering to spend time nudging pixels and changing values." Despite the fact that I am the only developer, I thought that this would be an excellent idea; it would allow me and my client to more easily experiment with the appearance of the app, and it would allow for easy setup of different "themes" for different companies: an element of the brief.

Q Branch had open-sourced the framework, so I implemented it with the app. Originally, the developers had read the themes off the XML property list file format. For the app thus far, I have been using JSON to store data, as it is easier to read and understand for people who may not have experience with XML formats. I therefore adapted the code so that it would read off a JSON file to better fit the purpose.

I tested the solution by trying to change the app's tint colour. By default, all app's use a blue tint colour equivalent to `RGB(0.0, 0.5, 1.0)`, but by changing the window's tint colour, the entire app can be themed in a particular way. In the theme JSON file, this is achieved by changing the line:

```
"AppTintColor": "ff0000"
```

This can be extended to appearance throughout the app.



Final Client Meeting

With the sole exception of the interface for the reporting functionality, the application is ready for use once the data for the assessments has been entered. To check the viability of the final product, I organised a meeting with two of my stakeholders: my client and one of his clients. There, I ran through the features and UI of the app, then gave it to my client to test.

Overall, the trainer was very pleased with the result. Any concerns which I might have had - such as the fact that the app requires Contacts integration in its current form - he thought of as only positives, saying that a trainer wants the information about their clients to be as easily accessible as possible, and he didn't conceive of a situation where a trainer might not want the functionality of their clients in their Contacts.

In terms of the appearance, he said that he liked the white background and the simplicity of the UI, describing it as "easier to read, more professional, and clinical," as fits an application designed to be used by professional trainers.

In testing the app out with his client, he realised that having a single screening form could be a potential issue; the app may need to store multiple screening forms, as, for example, a person could have a new injury that was not identified in the initial screening. However, he also wanted to maintain that the screening form was locked after it had been completed (with the exception of the first section, containing personal details); so, in the meeting, we decided to add an extra field to the first section of the form that could be edited at any time. This field contains the person's current goal or target, and will be displayed with the at-a-glance information on the overview screen. Having this allows the trainer to easily get an idea of what each client is working towards, helping them to better target their sessions.

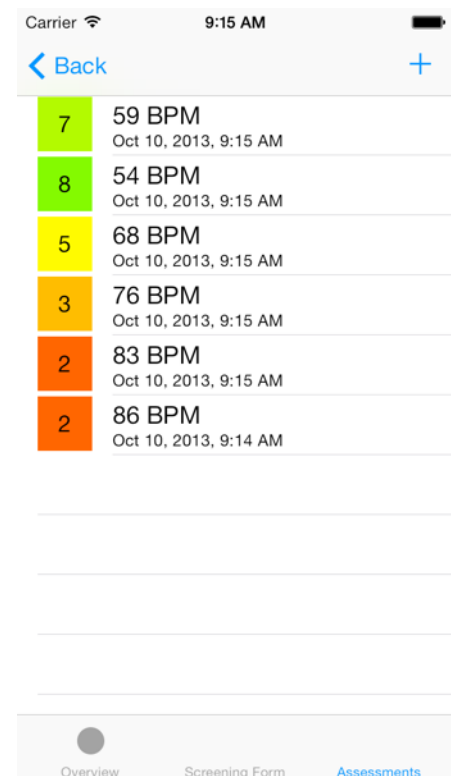
The trainer also noted that each assessment needs an 'Observations' field to feed through to the reporting functionality. This would allow the trainer to include information that a number cannot easily summarise, and which would be important for a clinical report; he said that while specialists tend to only see a client for a short amount of time, a trainer sees them on a regular basis, and as such any information that they collect is invaluable. This would be very simple to add.

When I showed him the exercise detail screen, which contained a graph of past results, he said that having the graph there was unnecessary and only cluttered the UI. The summary numbers provided good enough context for progress, along with the health wheel, that a trainer would not find the graphs particularly useful. He suggested to instead include them on the client report that would be emailed to a client after a session, as the client is the one who would more appreciate that sort of summary of progress. One thing he reinforced was that the user should be allowed to find their own workflow, and the app should not get in the way of that with complex features; if a trainer wants to see a graph for a client, they would just email themselves the client report. This was also referenced to in the context of the health wheel; I had contemplated including the ability to show progress over time, but he said that it would be better to just let people take screenshots of the graph for a particular time if they

wanted to save it; people would find their own ways to make things work, and more features are a hinderance, not a help.

The last minor change that came out of the meeting was bringing the result scores (the 0-10 assigned for each exercise result) throughout the app. He thought the visual indication here was extremely helpful, and should be shown before each exercise in a category and before each category, so the trainer can easily see what a person's weak areas are; having a red icon next to a category flags that as an area for attention, and when the trainer looks in that category, he or she would see what particular assessments were causing the low score.

In general, the client was very pleased with how easy it was to view information and use the app, and he could figure out the interface through very little experimentation. He was excited for the possibilities it holds for his business and his work as a trainer, saving him time and enabling him to better fit his clients' needs and the needs of the industry worldwide.



7	59 BPM Oct 10, 2013, 9:15 AM
8	54 BPM Oct 10, 2013, 9:15 AM
5	68 BPM Oct 10, 2013, 9:15 AM
3	76 BPM Oct 10, 2013, 9:15 AM
2	83 BPM Oct 10, 2013, 9:15 AM
2	86 BPM Oct 10, 2013, 9:14 AM

Overview Screening Form Assessments

Final Evaluation

I have developed a solution that is fit for the use of personal trainers in assessing their clients. The application meets the requirements in a number of ways:

- The interface is clean, easy to use, and visually clear. For tracking progress and entering data, this is incredibly important. Where useful, the application uses animation and dynamic hints to guide the user through it; the transparency effect and slide animation when a client is chosen is a key example of this, as it shows the user how to return to the previous view - when I tested this with a range of users, they could invariably find out how to return to the client selection screen within a short space of time. This is an example of the desired 'polish'.
- The codebase is compact and maintainable, for easy extension in future versions. It also represents a great development in my personal coding abilities and styles over my previous projects.
- The application is customisable and extensible, through my use of delocalised JSON files to theme the application and provide the data. It allows the application to be customised by other trainers so that it feels professional and polished for each.
- The application makes common data entry tasks easy; the elegance and ingenuity of a timer for correcting data vs. entering new data demonstrates an example of this.
- The application allows trainers to attach valuable data to individual results and have them automatically organised and formatted for client reports. This data includes notes/observations, numerical values, and photographic or video evidence.

- The conversion of a numerical result into a score provides visual summary as to a person's abilities, allowing the trainer to focus where it is most needed.
- Usability has been considered in the design of the app at every step, in enabling the user to view important information and perform common tasks with as few taps as possible, minimising 'friction'.

Reflection

The process of development has gone much quicker and smoother than I had expected; the decisions that I made during the brief development process helped me greatly in the prototype development. I knew it was an ambitious project, but in retrospect, using pre-existing solutions greatly expedited the process. Having a solid base to build on, using open source frameworks and the iOS SDK, allowed me to work on the features that mattered for my client, instead of having to reimplement a large amount of 'boilerplate' code.

Despite this, I was perhaps over-ambitious in the scope of the project, which has lead to time constraints for testing at this stage. I had hoped to have the trainers test the app extensively with their client, but due to the amount of time it has taken to implement all the features, I have been able to do less of that than I had hoped by this stage. In future projects, I may need to be more realistic in terms of scope to achieve a fully polished product in the allotted time frame.

Another success for this project, and an improvement over previous projects, is the extent to which I looked at others' technological practice to improve my own. Starting at the brief development stage, the insight I gained from other technologists' design and implementation helped me to gain clarity of how to achieve the outcome. For example, the way that DaisyDisk used the adapted pie graphs helped me to understand what made them useful in displaying information, and it meant that when my client described it, I could immediately get an idea of a possible implementation.

Through doing this app, I've learned a great deal in terms of both programming and design. The drastic changes from previous efforts in my coding/program design, prompted by looking at other people's styles, represent an evolution for the better; I've been able to accomplish more than I had expected in the timeframe due to how much less code I've had to write. Likewise, the design style, prompted by Apple's changes in iOS 7 and other peoples' apps (including user feedback on those apps), caused me to go for a much simpler look than I originally had in mind when I first heard the client's idea.

I believe I've shown ingenuity and creativity in some of my solutions to issues; for example, in the use of transparency and animation to guide the user through the application, and the way in which I utilised key-value coding to allow the information stored about a trainer's client to be more varied.

I've tried to prioritise the user experience throughout to result in a polished application, and this usually resulted in a more simplified outcome than I had originally imagined. Stripping away extraneous UI is a key example of this, as is using colour instead of large amounts of text for visual indicators.

Conclusion

The client was extremely pleased with the end result, and while I'm certain that there will be further minor alterations as my client and his colleagues use the apps in their work long-term, I'm confident that it provides a marked improvement over their past processes and enables them to better accomplish their work.

When the app is released to the broader market of the App Store, I know that users there will also feedback provided through the medium of reviews, allowing continual improvement for the application. No piece of software should be static; software should evolve with its audience. As it stands, the application is a successful evolution for trainers, bringing their existing systems into a digital age and making improvements for both themselves and their customers.

