iACT

MSc SD Project Implementation Report

Implementation of a mobile system for augmenting an ACT based psychological therapy process.

A dissertation presented in part fulfillment of the requirements of the Degree of MSc in Software Development at the University of Glasgow.

Giovanni Paolo Coia

1105874c

September 2012

Acknowledgements

I would like to express my thanks to all those who offered help, encouragement and advice during the completion of this dissertation. In particular, I would like to express my gratitude to the project supervisor, Dr. Joemon Jose, as well as the client for this project, Dr. Ross White.

Summary

This dissertation outlines the implementation of an iPhone and web based software solution for augmenting an Acceptance and Commitment Therapy (ACT) process. ACT is a form of psychological therapy that has proven effective in the reduction of depressive symptoms. The software is intended for use by both ACT therapists and their clients, and aims to help the client record thoughts or emotional events that arise during the user's day-to-day life.

iACT achieves this goal through the creation of an iPhone application permitting the user to record and explore thoughts in ways which are consistent with ACT's core principles. This aids the user in dealing with the content of difficult thoughts and allows them to keep a record of them. These records are then transmitted by means of an API to a web server. The user's therapist can then access this data by means of a web application to help them deliver a more effective therapy process.

The software is intended for use both during and between therapy sessions. In this way, it is hoped that the skills ACT aims to encourage users to exercise can be easily remembered and repeated. The project was built in response to the request of an academic, Dr. Ross White, who has research interests in ACT. Following an exhaustive requirements gathering process, this implementation process has delivered a fully functional system that meets almost all of his original objectives.

Table of Contents

1. Introduction

This implementation report outlines the development of iACT, a combined web site and iPhone application that together aid in the delivery of an Acceptance and Commitment Therapy (ACT) process. ACT is a form of cognitive behavior therapy that can be used to help therapist's clients, known as service users, to deal with the content of difficult thoughts. ACT is traditionally delivered by means of one on one or group therapy sessions. The primary goal of iACT is to deliver a tool that can deliver elements of an ACT process by means of an iPhone application, augmenting the traditional methods of delivery.

ACT's focus is on helping individuals live better, more meaningful lives. This is achieved by means of several core principles that ACT encourages service users to follow. The aim of these is to help the indivudual live with, rather than avoid, the content of their difficult thoughts. ACT teaches a technique, known as cognitive diffusion, to help achieve this. Cognitive diffusion is whereby the context of the thought is changed, and by doing so the hurt or pain associated with that thought can be reduced. Although there is no precedent for delivering ACT as a smartphone application, ACT has been successfully adapted to suit unconventional methods of delivery in other studies.

To achieve the aims of assisting the ACT process, the iPhone application will enable service users to record the content of difficult thoughts when they arise, then manipulate the thought through interactions based upon ACT principles. An issue commonly encountered by ACT therapists is that service users often struggle to accurately recall relevant emotional events during a therapy session. iACT aims to assist service users in their recall of events by also providing them with a thought history that can be discussed at meetings with their therapist. To permit the system to be used efficiently by the service user's therapist, a web application will work in conjunction with the iPhone software. When a user records a thought, this is submitted to the web application. This allows the therapist to easily view the user's thought history. Additionally, the therapist can add in extra thoughts that then sync back to the user's handset.

1.1 Problem/Motivation

A software solution to provide a mobile tool based upon ACT for use both during and between therapy sessions by service users was requested by the client for this project, Dr. Ross White. Dr. White is an academic at the University of Glasgow with research interests in ACT who had approached the Department of Computing Science with the original idea for the system. The software will help the user to understand ACT

and apply its methods and strategies to difficult thoughts they encounter in their lives. To best provide a software solution that the user would have readily available it was determined that the software would run as a mobile phone application, now known as iACT.

The client additionally wished for this mobile application to operate in tandem with a web server performing several functions. This server would collect usage data to allow therapists to assess their client's difficult thought history. This data should also be extractable in an anonymous form to permit studies of iACT's effectiveness should the system ultimately be used in trials. The thought history accessed through the site would be of particular value during therapy sessions, whereby the therapist could explore this with their service users. If, based on discussions with the service user, the therapist determines there is a certain thought the user has not noticed, the therapist can add the thought to their history and this is then synced back to the user's mobile phone. This means that the user can now be reminded to record incidences of this new thought.

The client had few requests as to how this solution be implemented. His only stipulation was that the application run as an iPhone application. In my proposal I gave some consideration to other mobile platforms such as Android and Blackberry. Although these other platforms would be valid candidates for an ACT application, they typically do not reach as many users as the iPhone app store, or suffer from extreme hardware and software fragmentation. This made the iPhone attractive as an initial platform for the construction of the system.

1.2 Implementation Objectives

Following the discussions with the client a number of objectives were outlined for the software. Although iACT would not be used as a research tool or be trialed during the development of this project, it is intended that the project produce software that could be used in the future by the client. The objectives of the software implementation were agreed as shown in *figure 1*.

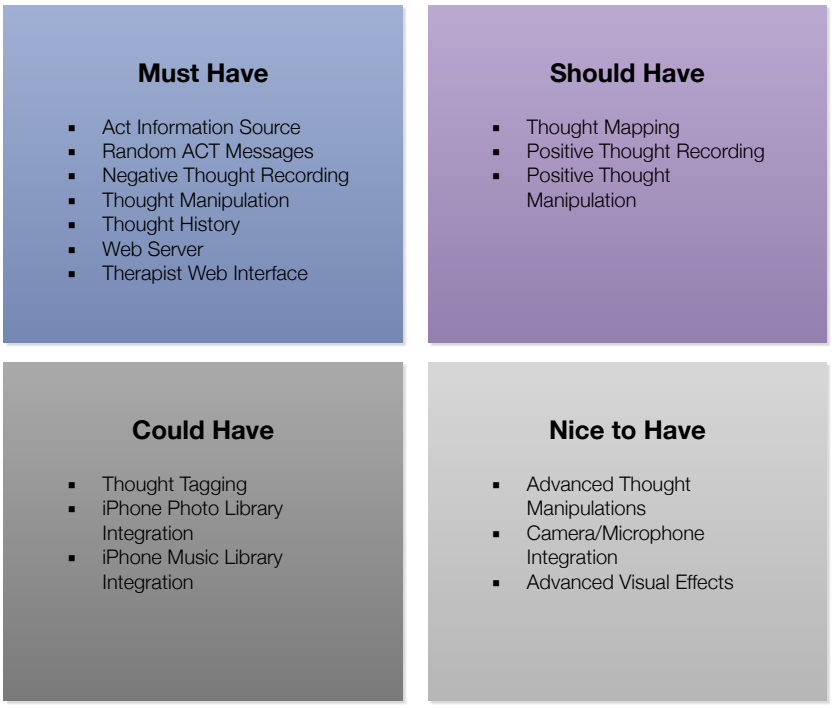


Figure 1 MoSCoW Software Requirements Chart

The chart above illustrates the requirements of the software. It was intended that I first ensure that the 'must have' functionality was implemented, before considering 'should have', 'could have' and 'nice to have' features in that order. This section explains in more detail the requirements of the 'must have' features.

1.2.1 Act Information Source

iACT should provide an information resource to its users that helps them to find out more about ACT and its core principles. This should be shown primarily in the form of an interactive hexagon diagram demonstrating the ACT model. Tapping on the various points of the hexagon should provide further information about that particular element. These six different principles, represented by vertices on the hexagon, form the key components of the therapy.

1.2.2 Randomly Timed ACT Reinforcement Messages

At random iACT should receive messages that appear on the user's iPhone. These messages will reinforce individual ACT principles and will take the form of iOS push notifications. These messages can be sent at intervals chosen by the user from a web server. The messages then appear on the user's iPhone in a similar fashion to a text message. This means they have a visual cue to indicate their arrival, along with an audible or vibrating alert.

1.2.3 Thought Recording

The user of the iPhone application should be able to record the content of difficult thoughts as they arise. These thoughts should be saved along with data such as location, time and rating. This data will then be transmitted back to the web application for analysis by the service user's therapist, or for research purposes to ascertain iACT's effectiveness.

1.2.4 Exploration of Thought with Interactive Graphics

Upon selecting this functionality in iACT, the user should be able to record a thought in the application as a short textual description. The application will then display the user's thought as a text string on the display. The iACT user should be presented with a number of tools to manipulate the textual representation of the thought on screen. The types of interaction offered could vary immensely, for example incorporating use of the iPhone's camera to capture images to use in the manipulations, or allowing the user to alter the size and position of the thought on the screen. The principle behind this feature is to encourage the user to alter the context in which they view these thoughts, to examine them in new and different ways. In doing so it is hoped that the user can exercise the mindfulness and acceptance skills that are a key part of ACT.

When recording a thought in iACT, the user will initially be asked to record the emotional impact of the thought. This could be achieved by asking the user to rate the impact on a sliding scale from most negative to most positive. After manipulating and 'diffusing' the thought the user will be asked to re-record the level of emotional impact. This before and after leveling will provide valuable feedback as to iACT's effectiveness in delivering an ACT based experience. One would hope to observe a positive change after the manipulation has taken place.

1.2.5 Thought History

When adding a thought a scrollable list of previously entered thoughts should be presented to the user. If it is a repeat occurrence of a past thought the user should be able to create a new instance of the previously entered thought. The application can monitor as an additional measure how many times a thought has been revisited. At a glance the application should display the thought description and an indicator describing how distressing the recorded thought was.

Such a feature is also intended to be advantageous during therapy sessions, with the user able to quickly call up examples of relevant thoughts or feeling as appropriate. Additionally, the therapist should be able to add examples of negative thoughts to the list so that the user can then quickly access that thought from the thought history to facilitate ease of use of the iACT application in-between therapy sessions.

1.2.6 Therapist Web Interface

All data collected by the iPhone application should be accessible by the service user's therapist. Using the web interface, the therapist can add new service user accounts to the system. The therapist can also add extra thoughts to their service user's thought history that then sync back to the mobile application.

1.3 Methodology and Approach

To deliver the project for the client I adopted a structured software development approach. This approach iterated through several stages. Firstly, the client's precise needs were established during a requirements analysis phase. This primarily took the form of extensive interviews. Once the requirements had been established and documented I proceeded to implement the proposed solution. After the solution had been constructed a detailed testing and evaluation phase followed. Changes were then made to the software following feedback from this process.

1.3.1 Requirements Analysis

For several months I met with the client weekly to discuss his requirements. At each of these meetings we explored the system's intended features. During these meetings we would discuss elements of the system, putting forth suggestions until we were both in agreement about the best way to proceed. From this I was able to draw up a detailed proposal along with UML charts such as use case and activity diagrams. These were coupled with prototype interface designs to ensure that what would ultimately be delivered met the client's needs accurately.

1.3.2 Implementation Process

Following the requirements gathering phase I then began the implementation. As before, I continued to meet regularly with the client, constantly keeping him appraised of the progress being made and taking on any feedback he had to offer. The detailed diagrams that had been developed during the requirements analysis phase formed a solid basis on which to begin development work.

As the implementation progressed the original requirements were refined further as I encountered issues or obstacles to progress. Some features were improved beyond the original specifications, whilst a few had to be scaled back. Once a working system had been established I proceeded to test the solution.

1.3.3 Testing

As the project entailed the development of two separate software products I adopted a separate testing strategy for each. For the web application I used a test-first methodology, writing test scripts for proposed functionality then working on the code until the test passed. This ultimately ensured that I delivered the required functionality and left me with a suite of unit tests that could be reused. As new features were added the suite of tests was constantly rerun to ensure new functions did not break already implemented features.

For the iPhone application a 'black-box' testing methodology was used. This is whereby each feature is checked at runtime to ensure it works as intended, using test data that explores the limits of the software. As functionality was added to the code versions of the software would be built to continually test existing functionality as well as the new.

1.3.4 Evaluation

The final stage of my approach was to evaluate the solution I had constructed. During the proposal phase it was agreed that iACT would not be trialed with actual service users at this time. This was largely due to the complex ethical and security issues that surround working with real users in a healthcare environment. To evaluate my solution I therefore reexamined the original objectives for the software and critically assessed how my finished products achieve these aims. I also conducted a small degree of user testing to ensure that the web application worked successfully with multiple users submitting data.

1.4 Main Achievements

The main achievements of this project are as follows:

- Successful development and implementation of a mobile system for ACT based thought recording and manipulation with the following features:
  - User log-in validation.
  - Thought recording.
  - Though History.
  - ACT information source with interactive ACT 'hexaflex' diagram.
  - User interface conforming to standard iOS design conventions.
  - GPS based thought location recording.
  - Exploration of thoughts with interactive graphics.
  - Sophisticated data model to record usage.
  - Fast multi-threaded REST HTTP request system.
  - Fully documented design.
- Successful development of an API and server system to capture and transmit data recorded by the mobile system with the following features:
  - REST based implementation.
  - Compatibility with other web services or devices created in the future.
  - N-Tier architecture.
  - Fully documented design.
- Successful deployment of a web application with the following features:
  - Ruby on Rails server.
  - Web-based interface for ACT therapists.
  - Tiered levels of access granting different functionality to different user classes.
  - User authentication system to facilitate access control on the web interface.
  - Model View Controller architecture.
  - Easily scalable.
  - Access to data recorded by iPhone application users.
  - Input validation with feedback messages.
  - Fully documented design.

Over the course of the project's development I have been able to construct a complete and fully functional system to augment an ACT process. Following the requirements shown in *figure 1*, I have successfully implemented all of the 'must have features' with the sole exception of a push notification service for the sending of random ACT messages. I was also additionally able to fully implement the 'should have' features as well.

1.5 Structure of the Report

This implementation report has been structured around several sections. Chapter 2 provides background to the project with an analysis of ACT and existing software in this field. Chapter 3 discusses the design process behind the system. Chapter 4 then outlines the implementation and I provide justification for the choices made in the development of the software. There is also an overview of the software testing approach that was used. Following both chapters 3 and 4 are design diagrams illustrating the functionality of the software. Chapter 5 evaluates the finished software against the original requirements. Some data captured from user surveys is also presented. Finally, chapter 6 provides an overview of future work that could be carried out, as well as my conclusions.

2. Background

In order to ensure I delivered an effective solution it was necessary to perform some background research, both into the ACT process and existing software products. Given the inter-disciplinary nature of this software development project, encompassing both technical elements and clinical therapy, and the need to satisfy the client's requirements, I performed a great deal of background reading. Although iACT is presumed to be the first time a smartphone application has been built to assist in the delivery of ACT, there was a handful of existing software products with similar aims or implementations. I was also able to exploit the client's expertise in ACT to learn a great deal during our meetings.

2.1 Therapy and Psychology

To ensure I created a system that would be of use to ACT therapists I undertook some research into the therapy process itself. Additionally, I considered if ACT could be effective when delivered as a smartphone application rather than the traditional individual or group therapy environments in which it can normally be found. I also attended a workshop on ACT delivered by the client during which he gave some practical demonstrations of using ACT. I also studied some of the key papers published in the field. Following this I was able to gain an understanding of the therapy process.

2.1.1 ACT Overview

ACT is a form of psychological therapy. An abbreviation of "Acceptance and Commitment Therapy", its goal is "to create a rich and meaningful life, while accepting the pain that inevitably goes with it"[1]. It is concerned with "taking effective action guided by our deepest values and in which we are fully present and engaged... it is only through mindful action that we can create a meaningful life"[2]. In our attempts to live a 'meaningful life' we of course routinely encounter negative or unwanted feelings; ACT helps to deal with these by teaching 'mindfulness skills' to help overcome them.

Rather than focus on getting rid of negative thoughts, ACT instead encourages individuals to examine and understand them. ACT posits that language is the source of much of the negative feelings we experience. By understanding our thoughts in the form of words our minds are setup in a manner that often encourages 'experiential avoidance'. This encourages us to treat negative thoughts as problems to be solved, typically devising solutions or strategies using problem-solving skills that are heavily rooted in lingual syntax. These solutions are often not solutions at all. Instead they are merely strategies to avoid that feeling.

ACT then, rather than try to mitigate or remove these thoughts, focuses on encouraging individuals to accept them. By spending less time creating 'experiential avoidance'[3] one can focus on concrete ways to improve wellbeing. These are the 'committed actions' that individuals can make. ACT does this in accordance with its six core principles, as shown in *figure 2*. This shows the principles laid out in the ACT 'hexaflex', a common device used to explain ACT during therapy sessions.

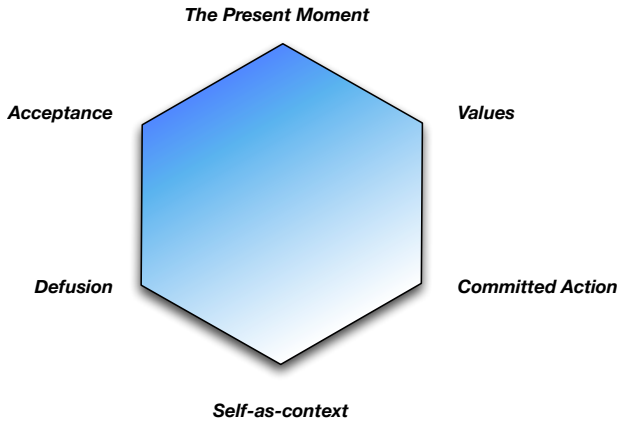


Figure 2 ACT Hexaflex

ACT's popularity has increased in recent years, prompting a number of studies into its effectiveness in treating depressive symptoms. These typically consist of a comparison of two groups of service users undergoing treatment for depression. The first is typically a control group whilst the second is a group receiving ACT based therapy. One such study[4] considered the effect of providing early interventions based on ACT to a group on a waiting list for treatment whilst the control group received none. The research demonstrated large reductions in depressive symptoms in the group receiving the ACT interventions. One could certainly imagine such early interventions taking advantage of a mobile application to gain further benefits.

2.1.2 ACT as Mobile Software

Although to both this author's and the client's knowledge ACT has never been delivered by means of mobile software, there is some evidence to suggest that it may be suited to alternative means of delivery such as iACT. One study has found that the delivery of ACT by means of brief telephone conversations rather than face-to-face therapy sessions was still extremely effective.[5] Other research has also demonstrated that emotionally disturbed patients may often struggle to recall emotional events.[6] Often, a patient may recall "categories of events, rather than retrieving a single episode."[7] By encouraging the user to record the thought as it occurs the service user may be able to more accurately retell of difficult emotional events during therapy sessions.

A good example of how ACT could be adapted to fit an iPhone application is with regard to ACT's 'cognitive diffusion' principle. A key component of ACT's emphasis on mindfulness, the user is encouraged to try to accept the difficult feeling by taking a step back, to look beyond its simple literal expression. Rather than through a verbal exchange with a therapist, iACT will encourage the user to diffuse difficult thoughts through the use of visual manipulations that the user will be invited to create each time a thought is recorded. Dr. Russ Harris, a noted expert on ACT, writes: "In a state of cognitive fusion we are caught up in language. Our thoughts seem to be the literal truth, or rules that must be obeyed, or important events that require our full attention, or threatening events that we must get rid of... Cognitive defusion means we are able to 'step back' and observe language, without being caught up in it. We can recognise that our thoughts are nothing more or less than transient private events—an ever-changing stream of words, sounds and pictures. As we defuse our thoughts, they have much less impact and influence."[8] This is a concept that potentially could be explored in an engaging way on a touch screen based device such as the iPhone. The 'multitouch' interfaces present on

iPhone devices provide a variety of methods for exploring information and ideas through dynamic interactions with the user and rich multimedia. The user will be able to visually reconstruct the strict literal expression of their thought and in doing so hopefully gain a better understanding.

Harris himself alludes to a similar concept to iACT's thought manipulation feature in an example where he retells his experience of an ACT therapy session: "I had him bring to mind the thought 'I'm a loser'... I asked him to imagine the thought as words on a Karaoke screen; then change the font; then change the colour; then imagine a bouncing ball jumping from word to word. By this stage, Michael was chuckling at the very same thought that only a few minutes earlier had brought him to tears."[9] This is precisely the kind of interactions that could work well on an iPhone, and if implemented effectively could hopefully provide similar effects.

Similarly, the simple act of recording the thought in the application can help promote ACT's 'acceptance' and 'present moment principles' by encouraging the user not to ignore the thoughts and put them to one side. By letting the user record the thought it may encourage them to live in the present and not try to hide from the difficult or negative thought that is troubling them. The recording process may help the user to put them into context, allowing them to accept and live with the thought that has been troubling them.

In discussions with the client, it was mentioned that these mindfulness and acceptance skills are practiced effectively during the therapy sessions, but individuals can often forget about them once they leave. The iPhone application can help reinforce these skills and remind individuals to utilise the mindfulness techniques in their daily lives. The software could achieve this through the use of the random ACT notifications it would send to its users. Additionally, users will be able to use the same application in both the therapy sessions and their private lives, allowing them to easily repeat some of the methods used during a session.

There is also research that demonstrates that the use of software in psychological therapies can have unique benefits not found in other forms of delivery[10]. In one example the author explains that "Application logs showed clients opening the diary without making entries, and also opening the diary, and recording entry a significant period of time later. Both of these cases suggest that the clients take the time to view (and perhaps reflect upon) diary entries."[11] This kind of data can be valuable to help assess the user's moods or responses to the therapy process. This kind of detailed information is also unique to computer software, being difficult to replicate in other mediums. iACT has therefore the potential to deliver a valuable ACT based experience to its users.

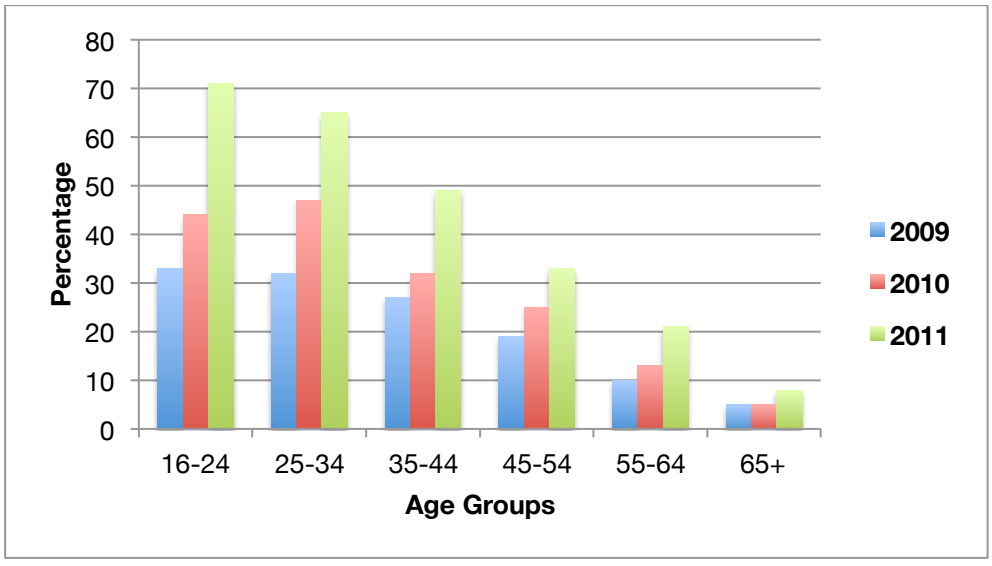


Figure 3 Percentage of UK Adults Using a Mobile Phone to Access the Internet

I also considered whether the technology iACT would rely on was likely to be mature enough for a typical person to have adopted it. Although mobile phone adoption is very widespread now, this does not necessarily dictate that these phones are being used to access the internet. The UK Office for National Statistics has performed a great deal of research into the Internet using habits of the UK public. This data, illustrated in *figure 3*, reveals that mobile phone based internet use has increased enormously in recent years, with a clear upwards trend. Perhaps not surprisingly the biggest gains are observed in the younger age brackets. Usage is however increasing across all age groups. The number of internet users accessing the web from a mobile phone increased from 23 percent in 2009 to 45 percent in 2011, representing some 17.6 million people[12]. This means a great many people are likely to have access to equipment that would allow them to take advantage of software such as iACT, and are comfortable interacting with their phones online.

2.2 Existing Mobile applications

At present there are few software products from which to draw inspiration. iACT is very likely the first time a software tool has been developed to augment the ACT process. Of currently available software, the closest in terms of function and implementation is an iPhone application for monitoring and recording the symptoms of Bi-Polar Disorder called *Bipol*[13]. There is also another application with similar implementation objectives, the *Poo Review*[14] application on the Android Marketplace. This slightly 'tongue in cheek' application asks users to gather data about their experiences in the lavatory to help in the treatment of bowel conditions. Much like iACT, numerical ratings and location data are gathered and then sent back to a web server. Although the motivation behind Poo Review may differ, its desire to collect data and submit it to a web server using a smartphone application to help in the treatment of a medical condition is identical to iACT.

2.2.1 Bipol

Bipol bears a number of similarities to the proposed iACT system in that it allows registered users to record a feeling state, as well as any important events that might have contributed to that state. It also allows the user to rate the state on a number of criteria, such as energy and anxiety. Bipol offers some similar functionality to the thought history feature of iACT, with the ability to view previously recorded moods and events.

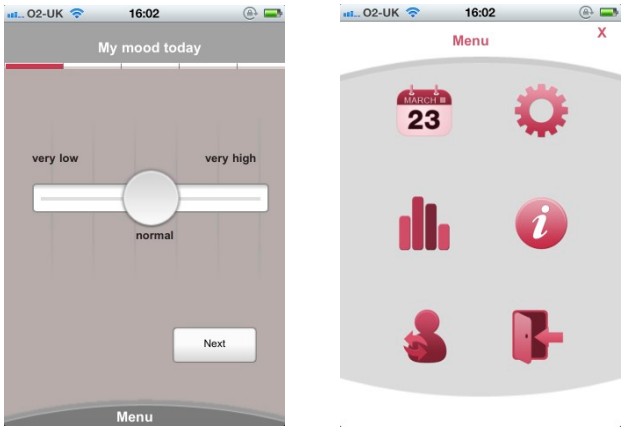


Figure 4 Bipol User Interface

The application also allows the user to view and manipulate the data in graph form permitting the user to observe trends or patterns in their moods. Like iACT, Bipol sends recorded moods back to a central website. On this site the user can establish 'Reviewers'; these are individuals who can log in and access the mood data the user has recorded in the Bipol application. A user can therefore choose to let a health professional or family member access the data. Again, this is similar to the therapist functions required on the iACT web site.

Bipol's intended users are very similar to iACT's in that both applications are intended to be easy to use for an iPhone user of any experience level and both seek to augment contact with therapists or health professionals. As such, some inspiration can certainly be drawn from its design and implementation. For example, the use of

simple sliders to input an indicator of present mood is something that I copied for use in iACT. There are however some drawbacks to Bipol's implementation. The user interface follows none of the typical iOS design conventions so it is not entirely intuitive to use. The buttons and menu structures are all unique to the application. Bipol is also simply a recording tool, it does not seek to enhance or implement elements of a particular therapy style in the way iACT incorporates elements of ACT. There are therefore no features similar to the thought manipulation or therapy information section in iACT.

2.2.2 Poo Review

Although Poo Review is not an existing mobile application to help deliver a psychological therapy process, the nature of its implementation made it valuable to consider. Poo Review is an application where users are asked to rate their experience in the lavatory. Similar to Bipol, instead of recording moods, stool is rated according to relevant criteria such as quantity and solidity. This data is then submitted back to a web server for analysis by a health care professional.

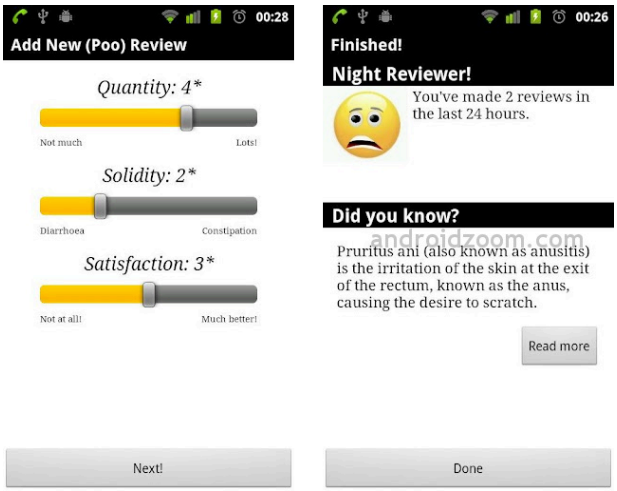


Figure 5 Poo Review User Interface

Poo review is a creation of Neal Lathia at the University of Cambridge who has usefully published a detailed paper on the implementation of his system.[15] Lathia's paper concerned itself with using mobile applications to monitor one's health. His goal was not explicitly to monitor bowel conditions, this was simply chosen as a good example to demonstrate his ideas. As such his paper examines a number of useful concepts for the implementation of any mobile health data gathering system.

Lathia notes that "mobile systems are becoming a bi-directional medium for data creation and exchange, which makes them perfect candidates for delivering tailored information or behavioural interventions" and that "a notable challenge in both domains is how to design these systems to enable collection of data while minimising the strain and interruptions that users must face."[16] Both of these observations are equally valid of iACT. To address these he argues that the application must be very quick to use and ideally fit in the user's idle time in the course of their day. In Lathia's case this was simple, the user was likely already idle in the bathroom at the time of the application's use. For iACT this is more challenging, given the application must be used during difficult emotional periods in the user's life. To this end I therefore regarded it as important that as much of the data entry be automated as possible, and that thought interactions take as little time as necessary. In iACT this could be done through ensuring signing in to the application only takes place once, capturing location data in the background and minimising the number of interactions required to record a single thought.

[1] Harris, R., 2006. Embracing Your Demons: an Overview of Acceptance and Commitment Therapy. *Psychotherapy in Australia*, 12(4), p.2

[2] Harris, R., 2006. Embracing Your Demons: an Overview of Acceptance and Commitment Therapy. *Psychotherapy in Australia*, 12(4), p.2.

[3] Harris, R., 2006. Embracing Your Demons: an Overview of Acceptance and Commitment Therapy. *Psychotherapy in Australia*, 12(4), p.2.

[4] Bohlmeijer, E. T., Fledderus, M., Rokx, T. A. J. J., & Pieterse, M. E., 2011. Efficacy of an early intervention based on acceptance and commitment therapy for adults with depressive symptomatology: Evaluation in a randomized controlled trial. *Behaviour Research and Therapy*, 49(1), p.62.

[5] Schimmel-Bristow, A., Bricker, J. B., & Comstock, B., 2012. Can Acceptance & Commitment Therapy be delivered with fidelity as a brief telephone-intervention? *Addictive Behaviours*, 37(4), p.517.

[6] Williams, J. M. G., Barnhofer, T., Crane, C., Herman, D., Raes, F., Watkins, E., & Dalgleish, T., 2007. Autobiographical memory specificity and emotional disorder. *Psychological Bulletin*, 133(1), p.122.

[7] Williams, J. M. G., Barnhofer, T., Crane, C., Herman, D., Raes, F., Watkins, E., & Dalgleish, T., 2007. Autobiographical memory specificity and emotional disorder. *Psychological Bulletin*, 133(1), p.122.

[8] Harris, R., 2006. Embracing Your Demons: an Overview of Acceptance and Commitment Therapy. *Psychotherapy in Australia*, 12(4), p.2.

[9] Harris, R., 2006. Embracing Your Demons: an Overview of Acceptance and Commitment Therapy. *Psychotherapy in Australia*, 12(4), p.2.

[10] Doherty, G., Coyle, D., & Matthews, M., 2010. Design and evaluation guidelines for mental health technologies. *Interacting with Computers*, 22(4), p.243.

[11] Doherty, G., Coyle, D., & Matthews, M., 2010. Design and evaluation guidelines for mental health technologies. *Interacting with Computers*, 22(4), p.243.

[12] Office for National Statistics, 2011. *Internet Access - Households and Individuals*. [Online] Available at: http://www.ons.gov.uk/ons/dcp171778_227158.pdf [Accessed 06 February 2012].

[13] Beating Bipolar, 2012. *Bipol-App*. [Online] Available at: https://www.beatingbipolar.org/bipol-app/ [Accessed 16 February 2012].

[14] Lathia, N., 2012. *The (Poo) Review*. [Online] Available at: https://play.google.com/store/apps/details?id=com.poo.review&hl=en [Accessed 20 May 2012].

[15] Lathia, N., 2011. *Using Idle Moments to Record Your Health via Mobile Applications*. Computer Laboratory, University of Cambridge.

[16] Lathia, N., 2011. *Using Idle Moments to Record Your Health via Mobile Applications*. Computer Laboratory, University of Cambridge.

3. Design

This chapter outlines my design process for the system. It examines my approach as well as providing an overview of the diagrams and documentation I created during this phase of development. For iACT a great many things had to be designed, including database schemas, activity diagrams, web site wire frames and APIs to facilitate the communication between the server and the iPhone. Following the creation of these I then considered which tools and technologies were most appropriate to begin the work on the implementation with.

3.1 Approach

The first stage in designing the iACT system was to establish exactly what the client required from the software. As iACT may one day be used in a real life therapy environment it was critical that I gain a solid understanding of the end user's needs. Over the course of the proposal and development periods I met regularly with the client to discuss the software. From these meetings I established detailed use cases and mock user interfaces. This ensured that both the client and myself were clear on what would I would be building. We also categorised proposed functionality according to its importance. This was to ensure that development time was spent on core functionality and not spurious extras. I would therefore focus on delivering the 'must have' features and only consider additional functionality if time permitted. This ensured I avoided the issue of 'gold plating'[17] the software, a common software development pitfall whereby far too much time is spent working on an ultimately unimportant feature.

When setting out to create iACT it rapidly became apparent that it was going to require the creation of a complex syncing system in order to facilitate the two way communications between the iPhone application and the web site. This required the construction of a tiered architecture, with the necessary software layers between the iPhone and the database to permit the exchange of data. It was also apparent that this would be the most difficult element of the system to implement, requiring the use of a large number of technologies as well as the complex debugging issues one can encounter in a tiered architecture. One of the first things I therefore considered following the initial system modeling was how other web sites approach the problem of syncing with remote applications.

Once I had established how synchronisation was going to work in principle, I then considered which technologies would be the best tools to implement the solution. Throughout the development process I was

keen where possible to adopt industry best practice using the latest methods and tools. Upon choosing the appropriate technologies I then set about building the web site and the iPhone application.

3.2 Modeling the System

3.2.1 UML

Having established the client's requirements through extensive interviews over a number of weeks, I drew up detailed UML (Unified Modeling Language) diagrams illustrating the proposed functionality. These can be found in the design diagrams section following this chapter. These allowed me to talk through proposed features, which, when combined with mock user interfaces, made it easy to understand exactly what I was going to deliver.

For iACT it was clear there would be two major user types for the system, the therapist and the service user. The therapist would interact with the system by means of a web application to view the service user's recorded thoughts and add new ones, while the service user would use an iPhone application to record and interact with these thoughts. For each of these users use case diagrams were developed to outline how they would use the software.

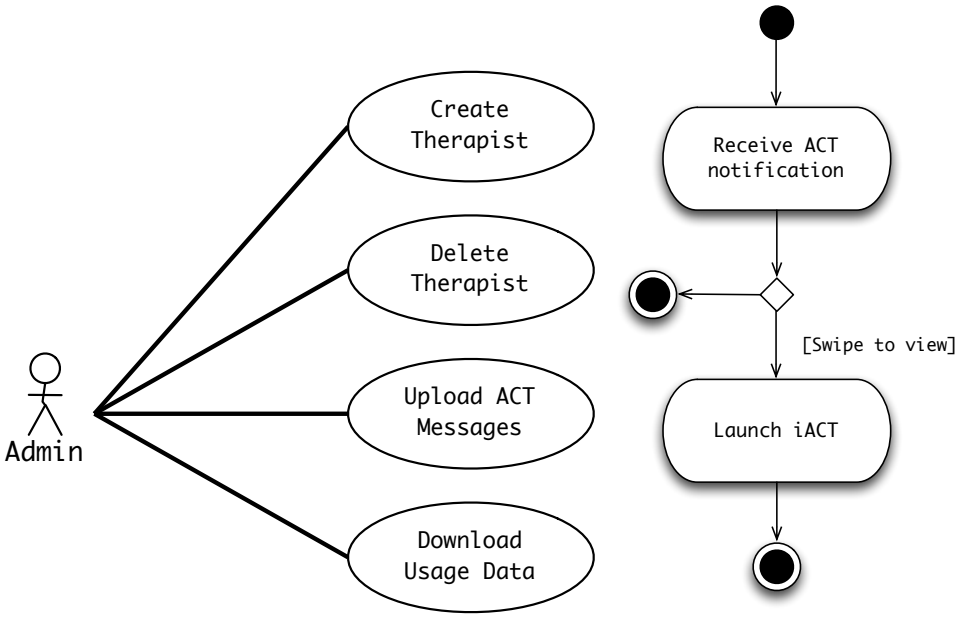


Figure 6 Use case and Activity Diagram Examples

Having established a solid list of use cases I then created activity diagrams for the key functions of the software. These are step-by-step walkthroughs that show exactly what each function will perform. Like the use case diagrams, these were used to ensure both I and the client had a clear understanding of precisely what would be delivered.

The final stage in this initial design process was the creation of detailed mock user interfaces. These implemented the use cases and activity diagrams I had developed and gave the client a good feel for what the end products would ultimately be like. These diagrams also then formed the framework on which I would begin developing the system.

As well as documenting the design in UML, I set about creating database schema for both the iOS application and the website. These diagrams would be utilised when I started implementing the data structures and helped underpin exactly what data the software would record. Once again this was done in consultation with the client, to ensure that the information that was going to be captured met his needs.

3.3 Online Infrastructure

As the design of the system's online infrastructure would ultimately dictate how I would set about developing the iOS application, I first considered how this would implemented.

3.3.1 Architecture

Given that the iACT system was going to require the synchronisation of small messages between an iPhone and a web site I examined existing web sites that already address this problem. The two I considered were *Foursquare*[18] and *Twitter*[19]. The former is a service for 'checking in' to public locations and alerting one's friends and the latter is a popular micro-blogging service. Both sites provide a public API (Application Programming Interface) to facilitate synchronisation with third party applications or clients. This has in turn allowed a large ecosystem of independently developed iPhone clients to emerge for both of these web services using these APIs.

Upon examining the APIs used in both of these services some common design themes emerged. Twitter and Foursquare both use a REST based API to send and receive data in the JavaScript Object Notation (JSON) format[20]. Further research into the matter quickly determined that the use of REST and JSON is now a widely adopted approach to solving such problems, as one is then left with a public API that can be used by any authorised third party client or service very easily. It also uses the HTTP protocol to transmit data, making it easy to incorporate into web servers.

3.3.2 REST and JSON

REST stands for "Representational State Transfer" and was created by Dr. Roy Fielding as part of his PHD dissertation.[21] It is in essence a set of rules for the architecture of web service APIs for simple CRUD (Create, Read, Update and Delete) interactions. One exposes the directory structure of the system through URLs that represent the structure of the data. For example, in iACT I was intending to model users who have many thoughts. I would therefore provide a URL along the lines of:

http://www.iact.com/users/1/thoughts/2

This URL would therefore return thought number two for user number one. In order to interact with a specific entity REST uses the basic HTTP methods with these URLs as explained in *figure 7*[22].

- To create a resource on the server, use POST.

```
POST /users
<user>
       <name>Test iACT User</name>
</user>
```

- To retrieve a resource, use GET.

```
GET /users/1/thoughts/2
```

- To change the state of a resource or to update it, use PUT.

```
PUT /users/1
<user>
       <name>Test iACT User</name>
</user>
```

- To remove or delete a resource, use DELETE.

```
DELETE /users/1
```

Figure 7 REST HTTP Verb Overview

Typically the information sent and received will be in an open interchangeable format such as XML or JSON. For iACT it became clear JSON would be the best option, for it is significantly easier to read and the iOS provides powerful built in tools to parse JSON into objective-C objects, the native programming language used by iOS software. Another advantage in using JSON is that outside of the raw data itself there is very little additional text. This means that JSON files are typically smaller than an XML file containing the same data. As the service will be used on mobile phones, ensuring the data transferred is as small as possible should greatly improve the application's performance in situations where the user has a poor mobile data connection. Some users may also be billed by their service provider according to the quantity of data they use. For these users, excessive data sizes may mean unnecessary charges on their phone bills. Lastly, using the mobile data connection can have a significant detrimental effect on a phone's battery life. Ensuring it is therefore used as little as possible was a design priority, and one that JSON was better suited to achieving than XML.

Before finally settling on REST based interactions I considered some of the alternatives. The other commonly used solution for designing APIs for web services is the Simple Object Access Protocol (SOAP)[23]. Similar to REST, SOAP is also an architecture for the retrieval of structured data and it too can use the HTTP protocol. SOAP is however entirely XML based, meaning it has the same drawbacks as using REST with XML. SOAP's real advantage is that is can be used with protocols other than HTTP. This was not a concern for iACT, and given SOAP's increased level of complexity and larger file sizes REST seemed a better choice. Having established that REST would form the basis of the interactions between the iPhone application and the server it was apparent that the architecture of the synchronisation system would look like that shown in *figure 8*.

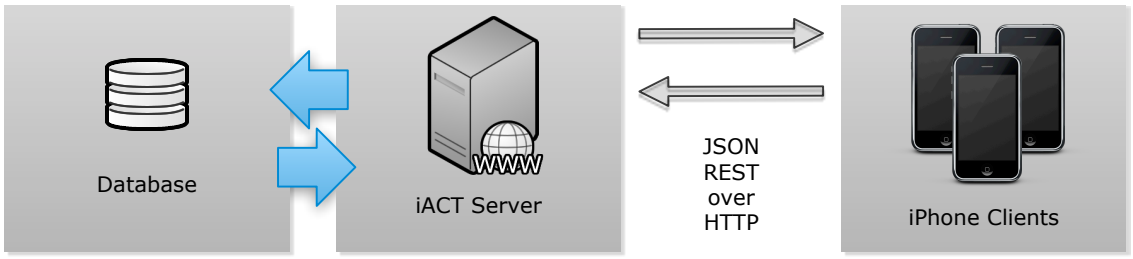


Figure 8. iACT Server Overview

With the online architecture settled I then considered the design of the iACT web application that would deliver the REST API, amongst other features.

3.4 The iACT Web Application

The first stage in designing any web application is typically to construct what are known as wireframes. These are detailed diagrams that outline a prototype user interface for the site. These are usually annotated to explain the function of each part of the website and form a good starting point from which to begin development. From these diagrams one can quickly understand exactly what functionality will ultimately need to be implemented. An example wireframe is shown in *figure 9*. The full set of wireframes I developed can be found in the design diagrams section at the end of this chapter.

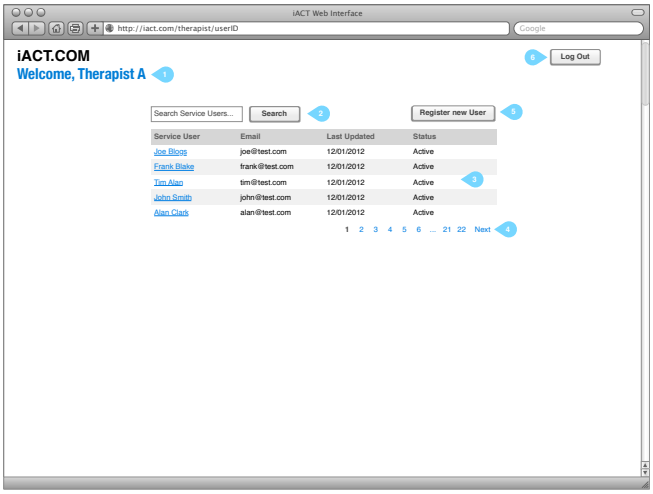


Figure 9 Wireframe Example

3.4.1 Web Application Architecture

Having now established that the primary means of interaction between my iPhone application and the website was to be a REST based API I set out to find the best tools for implementing my wireframes. I soon discovered that using a web application framework was likely to be the best option for the development of the system. As part of the design process I had outlined tiered architecture for the web application as shown in *figure 10*.

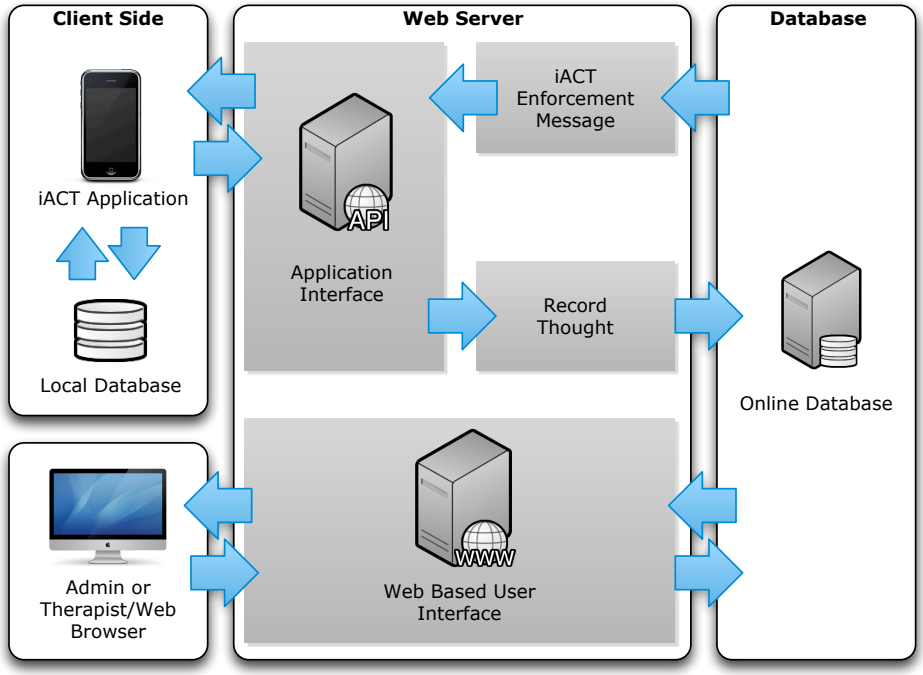


Figure 10 System Architecture

The above figure provides an overview of the three-tier nature of the system. In the client side tier is the web browser or iPhone application. The iPhone application additionally interacts with its own local database. The second tier of the system is the web server component. This provides the API for the iPhone software and a web based user interface for therapists accessing the system through a browser. The web server mediates interactions with the back end tier, illustrated here as the database tier. This tier will store the server side copies of the thought and user data captured by the client side software. The web application framework

would handle the web server and database tiers of the system. I had already established that REST would handle the API layer for the application interface.

3.4.2 Web Application Frameworks

Web application frameworks (WAFs) are software solutions that help in the development of dynamic web sites by providing ready-made libraries to support common web development tasks. For example, most frameworks will provide ready made database interaction and user authentication code. This means the developer spends considerably less time on so-called 'boiler plate' code and can concentrate on implementing their web application's main logic and design. WAFs also typically adhere to the model view controller architecture style allowing one to easily separate concerns in the design of their site. This makes modifications to the code in the future significantly less troublesome, as there is likely to be a very low degree of coupling in the code.

At present there is an enormous number of WAFs available to use, and the approach and design considerations of each can vary enormously. Not all WAFs will be suited to all types of tasks. In searching for one to use for the iACT website I had several key objectives, as outlined in *figure 11*.

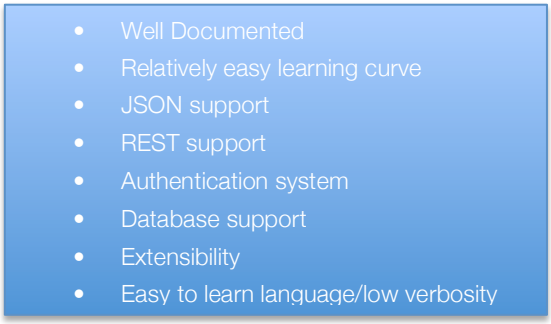


Figure 11 WAF Requirements

Although WAFs can make many development tasks easier, they do have some drawbacks. They often have a steep initial learning curve, as one must learn how to use the frameworks provided. This was doubly so for me, as not only would I have to learn how to use the framework, I was also going to have to learn fundamental aspects of web design such as HTML and CSS. Despite this, it was apparent that in the longer term adopting a good quality web solution from the outset would have several benefits. It would also save large amounts of time when implementing some of the more complex features the site would require. The classic draw back of using a WAF is that they are usually 'opinionated software'. This means they are often designed with a very strong opinion of what the 'right' way to do something is. If a WAF is used in a way for which it was not intended it can often become frustrating or difficult to work with. For iACT, however, I did not foresee this becoming an issue. This is because iACT's intended functionality is fairly standard for a dynamic web application.

Many WAFs also make interacting with SQL (Structured Query Language) based databases much easier by implementing an Object Relational Mapping System (ORM)[24]. ORM systems represent entities in the database using objects native to the framework's language, enabling the developer to work with databases very easily and without writing a single line of SQL. This also helps make the code significantly easier to write and to understand. The framework handles the SQL on the developer's behalf in the background. Another attractive feature of WAFs is that they are usually extensible. One can quickly add functionality for new features by installing a ready built software component.

In deciding what WAF to use I examined several different options such as *Django*[25], *Flask*[26], *Node.js*[27] and *Ruby on Rails*[28]. Given my requirements included the framework being based on an easy to learn programming language I quickly discounted PHP, Java and JavaScript based frameworks in favour of Python

and Ruby based ones. After some experimentation it became clear that the Ruby programming language was both easy to use and provided very human readable code, much more so than the PHP and Java based options. It also has none of the annoying indentation error issues that can quickly begin to annoy when working with Python. I also found it to be significantly better documented than the Python based WAFs such as Django and Flask.

The decision was ultimately very easy upon discovering that REST based design is at the core of how Ruby on Rails works and further, it has native libraries built in to display database records in the JSON format in response to a REST request. This makes Ruby on Rails an extremely good choice as the back end for any iPhone application, and it suited iACT's requirements well. Indeed this is the same solution adopted by some of the biggest web application/iPhone client systems on the Internet such as Twitter. Using Ruby on Rails would mean I would have to write much less code to facilitate JSON templates, authentication, database interactions and so forth.

3.5 The iPhone Application

With the designs of the web site and REST APIs decided I then examined my options for developing the iOS application. I consulted a range of online resources and textbooks to gain a good working knowledge of the tools available for iOS development. Once I had established the tools I would be using I then considered some of the choices I would need to make in the design of the software.

3.5.1 Production Environment

The range of development options for the iPhone application was vastly more limited in comparison to the web site. Although officially Apple's own Xcode IDE (Integrated Development Environment) is the only way to develop iOS applications, in practice there are two widely utilised development environments. These are *Xcode*[29] and *MonoTouch*[30].

MonoTouch is a third party IDE that has recently become quite popular for creating iOS applications. This is because unlike Xcode, it can create native applications for a wide range of platforms from the same code. All development in MonoTouch takes place in C# and uses the .NET libraries. MonoTouch then converts this to native code for use on the iPhone at compile time. Apple, however, does not support MonoTouch in any form and there has been some concern expressed that the iOS SDK terms and conditions may actually prohibit MonoTouch developed applications from reaching the App Store. This is in large due to a clause that used to be present in the iOS SDK (Software Development Kit) license agreement[31] that expressly forbade iOS applications being compiled from anything other than Apple's standard objective-C/C based code.[32]

The rules have subsequently been relaxed slightly, but Apple has often made substantial changes to the SDK license agreement with no notice. This means that any development that takes place outside of Apple recognised methods might well be at risk from rejection from the App-Store in the future. MonoTouch's real advantage is for developers who already have substantial experience with, or large code bases in, C# and .NET and wish to support a wide range of devices from the same code. For a new developer such as myself, this poses no advantage, particularly as this project is solely iOS based. Perhaps the biggest advantage of Xcode however is the range of books and documentation available. There are far more resources for working with Xcode than MonoTouch. Given these advantages, I elected to use Xcode to create the iACT application.

3.5.2 iOS Versions

Upon establishing that iACT would be developed for iOS devices using Xcode the next decision was to choose which version of the iPhone operating system to target. When developing iOS applications one of the first decisions the developer must make is which version of the platform to build the application for. This then determines which APIs and frameworks the developer can use, and then in turn determines which devices will ultimately be capable of running the software. With an application such as this intended to be useable by as large a subset of users as possible my initial intention was to try to target an older version of the iOS.

However, upon some research into the matter, I quickly determined that targeting the latest iOS, version 5, would actually bring a number of benefits both to me during development and to the project itself.

Firstly, iOS 5 was at the time of the development the latest publicly available version of the iOS. At the time of writing the iOS 6 beta is available for development purposes, but is not intended to launch until an undetermined date in the Autumn. iOS 6 is in comparison to previous upgrades relatively incremental[33]. In contrast to the release of iOS 5 there are few, if any, paradigm-shifting changes. It is also likely that an application targeting iOS 5 will run on an iOS 6 device with few or no modifications.

Secondly, iOS 5 was a major release in the history of the iOS introducing several changes that fundamentally alter how one approaches developing for the platform. For a developer new to the platform such as myself, it seemed backward to learn the older style of development, as this is not applicable to the later versions of iOS. Xcode also saw some major changes. For example, Interface Builder, the tool Apple provides to construct the GUI (Graphic User Interfaces), was in prior versions a stand-alone application. It is now fully integrated into Xcode itself. This means that textbooks teaching development for iOS 4 are hard to apply to iOS 5, and again I wanted to learn the latest in best practices for development.

The final factor in my decision to target iOS 5 was the distribution of iOS versions across all users of iOS devices. At Apple's World Wide Developer Conference in 2012[34] it was announced that of the 365 million iOS devices the company has sold, 80 percent of these are now running iOS 5. In short, iOS 5 adoption is now near uniform, and given the new style of development it requires it did not seem a worthwhile use of my limited time to try to learn how to support older versions.

3.5.3 iOS 5 and Memory Management

Another of the major changes heralded by the release of iOS 5 was a completely new approach to how memory is managed. My prior, and until the beginning of this project only, experience in programming was in developing Java software. In Java I was to discover one is somewhat spoiled in that memory management is not a concern of the programmer, being handled in its entirety by Java's garbage collection feature. Garbage collection is whereby objects stored in memory that are no longer required are removed automatically at regular intervals. There is no garbage collection in iOS. Until the release of iOS 5, developers had to manually manage memory through the use of 'retain', 'release' and 'autorelease' commands.[35]

These commands are necessary because iOS uses a technique known as reference counting to manage memory. The commands work as follows; "Each object has a reference count associated with it. When some part of an application takes ownership of an object, it increments the object's reference count by sending it a retain message. When it's done with the object, it decrements the reference count by sending a release message to the object. When an object's reference count is zero, it is deallocated."[36]

In order for this technique to function correctly it is imperative that the retain and release messages balance. Should a retain message be sent to an object more times than release is sent, the object will never be removed from memory with the resulting risk of memory leaks. Similarly, should release be sent more times than retain, a null pointer situation can arise whereby the object is no longer present at the address the code expects it to be. In practice it was a widely experienced problem for new iOS developers that balancing these messages quickly became one of the main difficulties in the learning process. Fortunately, iOS 5 has a solution for this problem called ARC.

3.5.4 Automatic Reference Counting (ARC)

One of the headline features of iOS 5 from a developer's perspective was therefore the introduction of Automatic Reference Counting. This almost completely removes the need for the developer to worry about managing memory. When coding for iOS 5 it is no longer necessary to worry about the retain, release and autorelease commands.

Alongside the release of iOS 5 a new updated version of Xcode was released, version 4. In this release Apple replaced the default GCC (GNU Compiler Collection) compiler with a new LLVM (Low Level Virtual Machine) one.[37] It is this new LLVM compiler that has enabled ARC. Whilst reference counting is still actually taking place, the compiler takes care of the work. The compiler additionally compiles twice as fast and produces more optimised code. The decision to support iOS 5 therefore has little or no drawbacks.

3.5.5 Storyboards

Alongside the introduction of ARC, the other major change in the development process is storyboards. New to iOS 5 and Xcode 4, they allow the developer to visually 'sketch' the interfaces for the program and the links between them. This allows you to walk through the 'flow' of an application visually in the IDE, and means one can very quickly get the interface up and running. This is not conceptually dissimilar, as the name suggests, to the storyboards that are often used to design movies.

3.6 Production Workflow

Having established the design for both my web application and my iPhone software the final consideration was to design the workflow approach I would use. I wanted to ensure that my approach would encounter as few problems or risks as possible. As Xcode is only available for the Mac, this dictated that I work on a Mac OS X based computer. This was not an issue, as this is the OS I use daily.

Developing in Xcode costs nothing, save for the cost of the Mac needed to run it. Apple makes Xcode freely available on the Mac App store. However, if one wishes to distribute their iOS creation on the App Store or test their code on a real handset, a developer account is required. This costs a small amount of money and grants you these additional abilities. As I ultimately intended to test the software it was necessary to get a developer account. The cost of admission also grants the developer access to Apple's extensive online documentation libraries, another advantage that I was keen to have.

The biggest concern of my workflow was the risk time lost due to data loss or computer malfunction. I adopted a wide range of strategies to mitigate this risk. Firstly, I used Mac OS X's built in 'Time Machine' feature. This performs hourly backups to an external drive, and also provides a degree of version control over your files as it allows the rollback of changes to a previous hour in the backup history. I also performed all of my work inside a DropBox[38] linked folder. DropBox is a web service that automatically backs the content of this folder to its own servers. The final element of my backup strategy was to use git[39] source control. I linked this a github[40] online repository for my software. This allowed me to easily perform version control as well as keep track of changes to my code. This four pronged back up strategy left me with two separate local copies of my work and two separate offsite backups that were never more than one hour out of sync with each other.

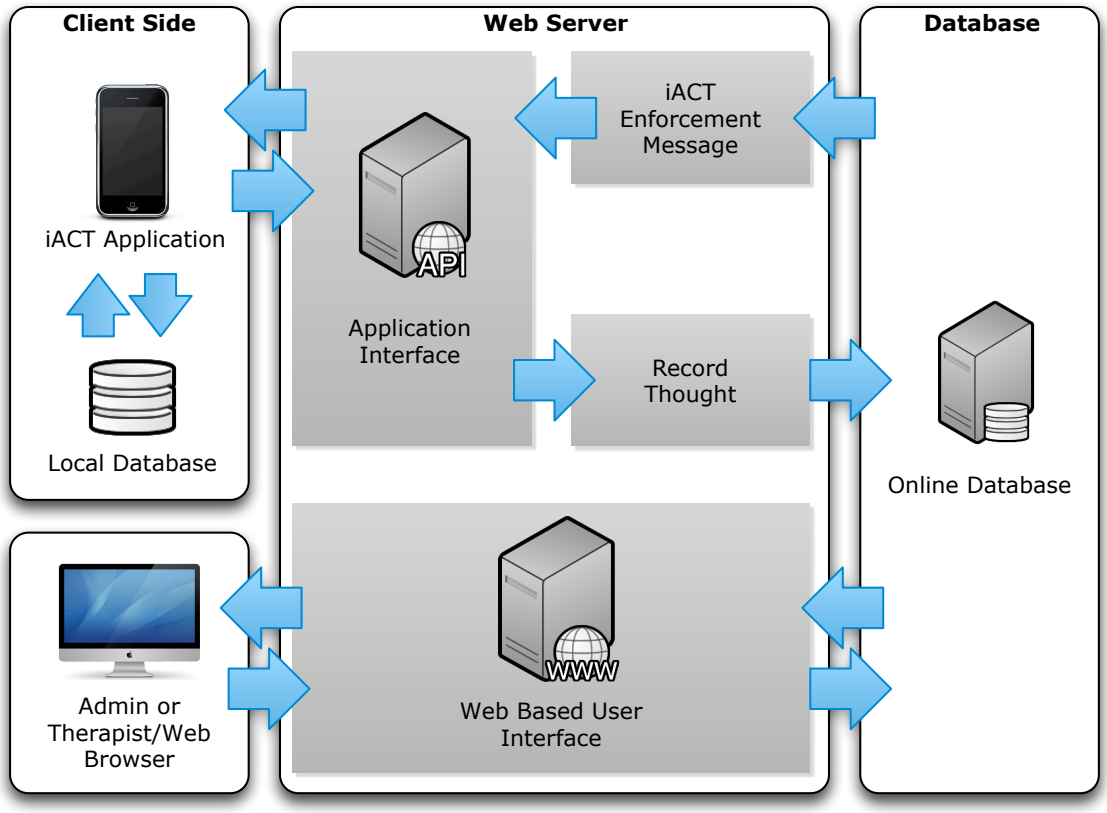
My final concern was the choice of a good quality text editor. Although most of the iOS work would take place within Xcode itself, large chunks of the development were not IDE specific. Ruby on Rails development can be performed in any text editor. To help myself I looked for a text-editing package that supported syntax highlighting and checking in a wide range of languages. To this end I purchased a copy of the excellent BBEdit[41]. This allowed me to easily work with the many different languages that would make up the iACT website, such as Ruby, HTML, CSS and so forth.

To manage my time effectively I decided to use a task management package called OmniFocus[42]. This enabled me to break down each development task into smaller manageable chunks, helping me to avoid becoming frustrated at the size of the task in hand. I therefore decided to work through the various development tasks as they occurred in OmniFocus using both Xcode and BBEdit to implement where appropriate. The backup strategy I had chosen meant I didn't have to worry beyond this.

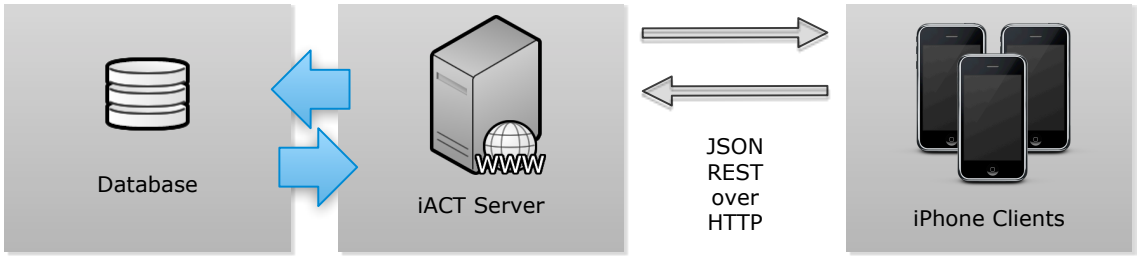
3.7 Design Diagrams

3.7.1 System Architecture

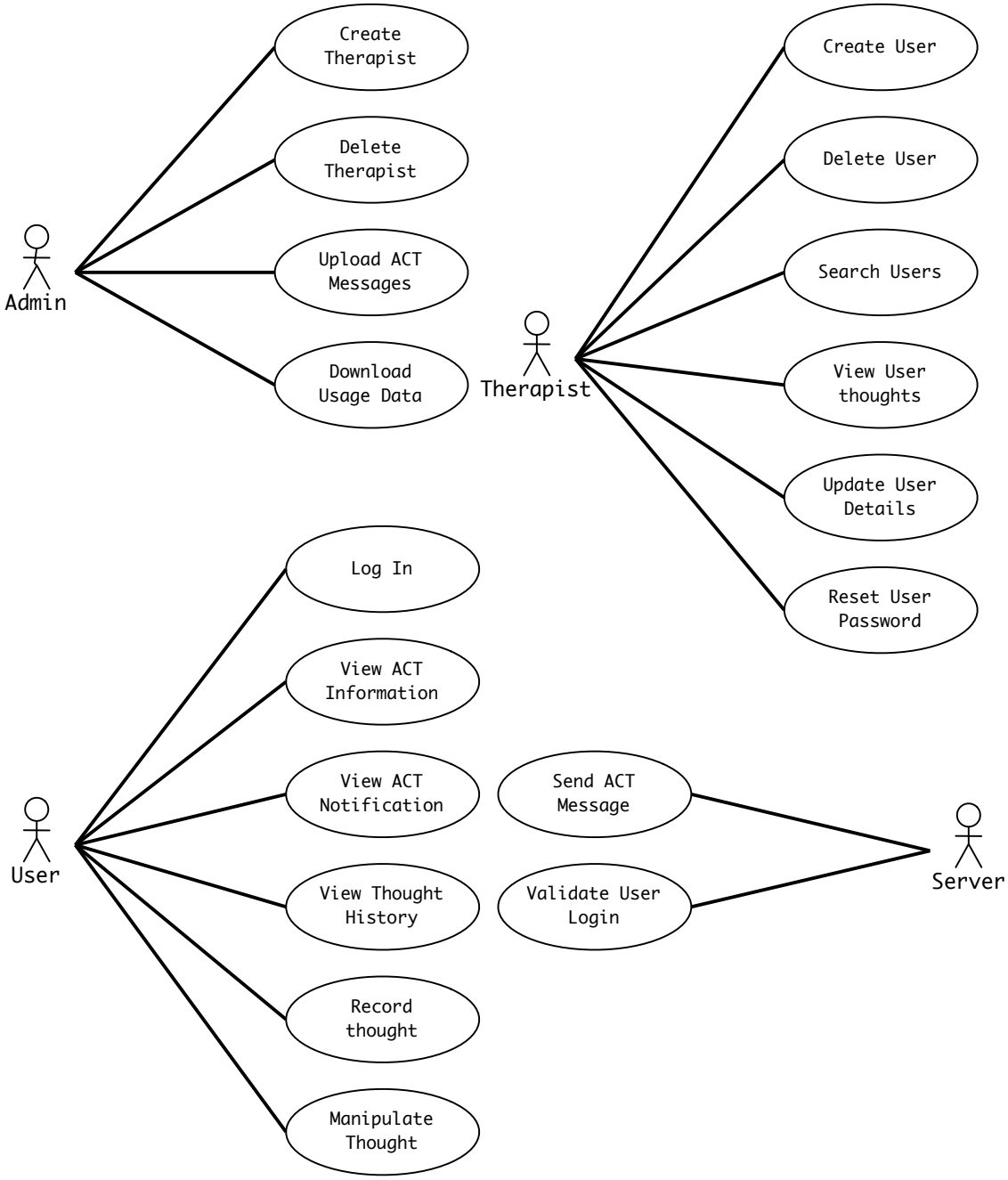
System Overview



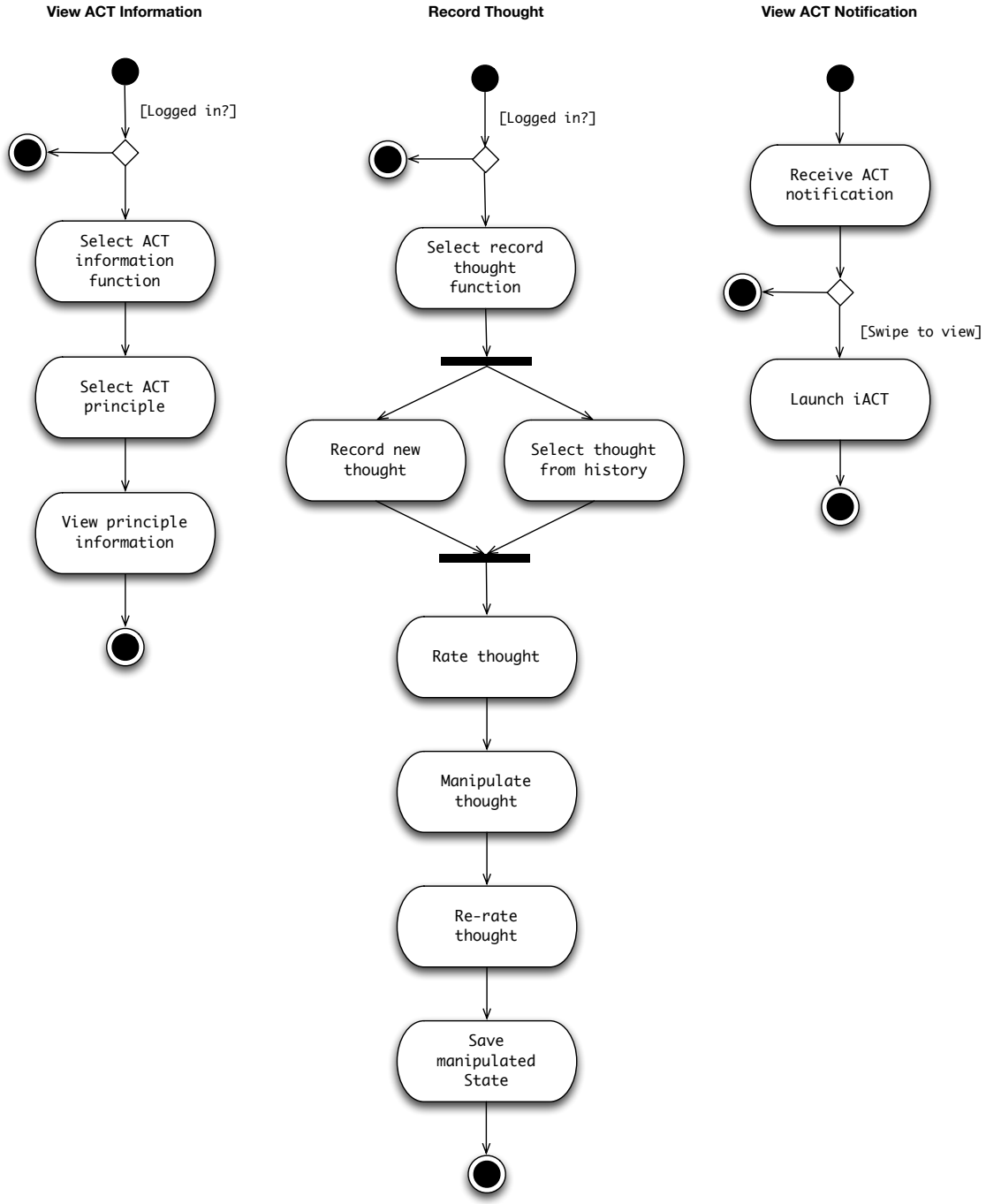
REST Architecture



3.7.2 Use Cases

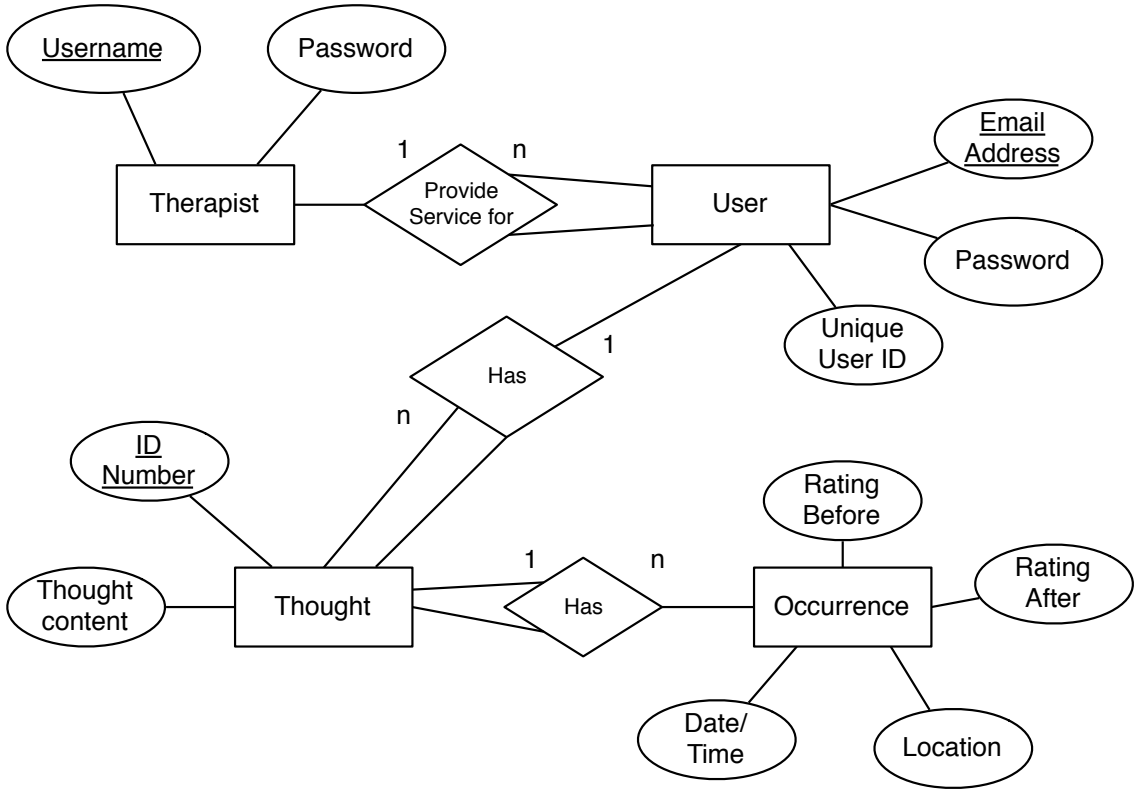


3.7.3 Activity Diagrams

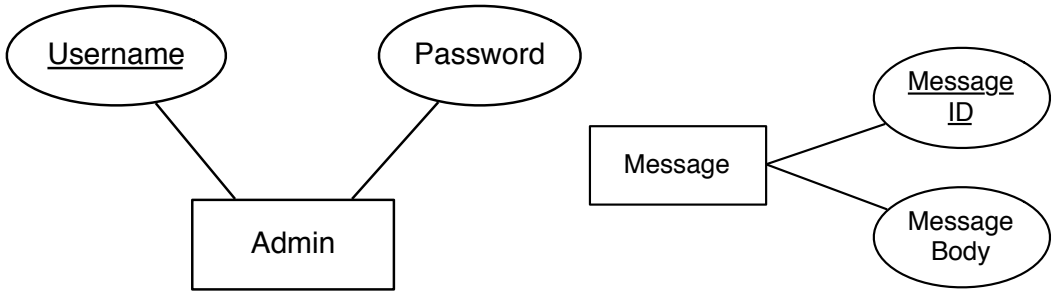


3.7.4 Database Schema

Primary Database Structure

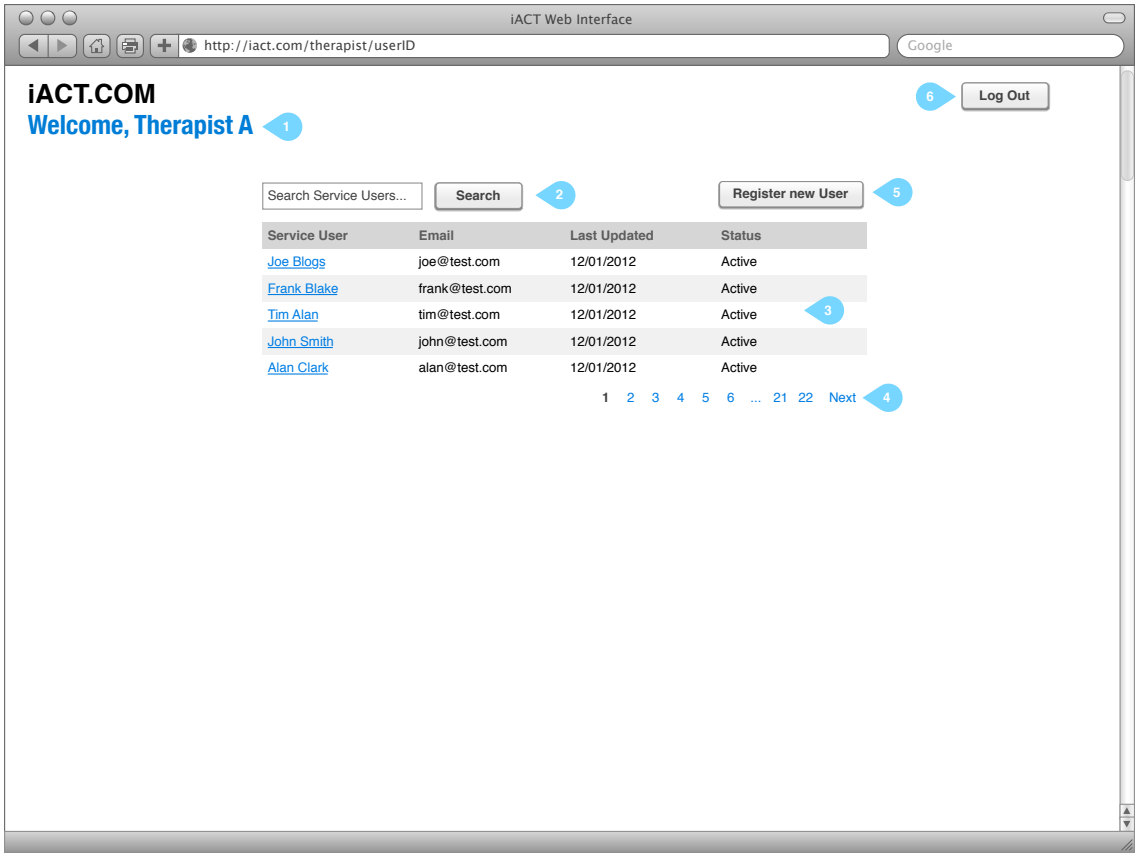


Admin Users and Push Notification Tables



3.7.5 Web Application Wireframes

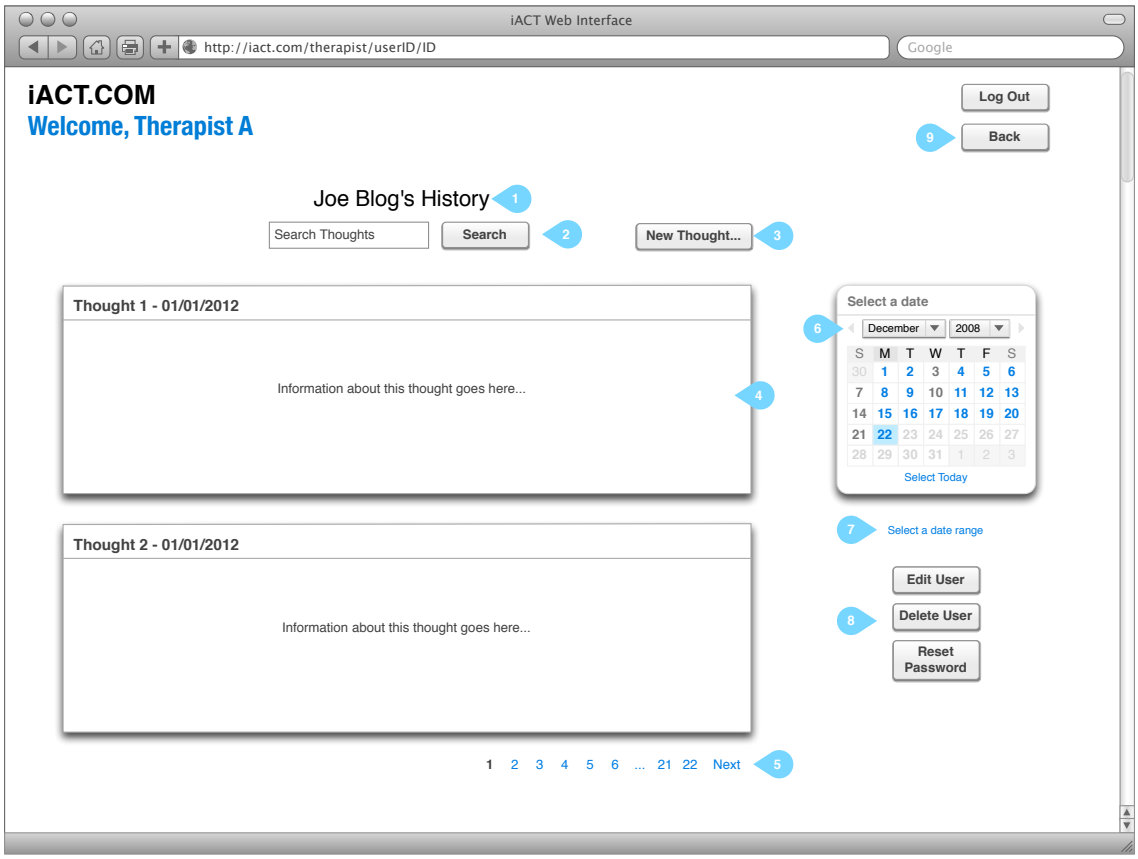
Main Home Page



When a therapist logs in a page providing similar functionality to the one above is displayed. Numbers correspond to the labels on the wireframe. The therapist can only search among service users they have personally added to the system. A therapist cannot see another therapist's clients, keeping their data secure.

1. Displays the name of the currently logged in therapist.
2. The therapist can search for service users.
3. Results are displayed here. If no search performed, shows the most recent activity. Clicking a service user's name will load the relevant service user's information page.
4. Results may spill across multiple pages.
5. If service user is not yet present on the system the therapist can register a new user.
6. When a therapist has finished looking for data they can log out.

Service User Page

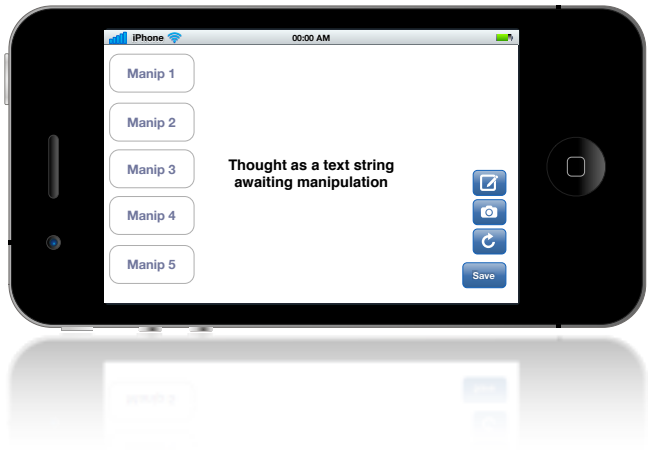


This page displays the information held about a given service user, and is shown when a service user is selected on the main page.

1. Displays the name of the person whose thought history is being viewed.
2. The therapist can use the search box to find specific thoughts...
3. ...or add a new thought to the list. This thought will then sync back to iACT on the user's handset.
4. Results are displayed here. If no search is performed, most recent thoughts are displayed.
5. Results may spill across multiple pages.
6. The therapist can filter results by date...
7. ...or select a date range.
8. These buttons allow the therapist to edit the user, for example changing the spelling of their name or adding a new email address. The therapist can also delete the user from the system or reset their password.
9. Clicking back returns the therapist to the main page.

3.7.6 Mock User Interfaces

Main Menu, Thought History, Record Thought and Push Notification Views.



Manipulate thought View.

---

[17] Wikipedia, 2012. *Gold plating (software engineering)*. [Online] Available at: http://en.wikipedia.org/wiki/Gold_plating_(software_engineering) [Accessed 19 August 2012].

[18] foursquare, 2012. *foursquare*. [Online] Available at: http://www.foursquare.com [Accessed 03 June 2012].

[19] twitter, 2012. *twitter*. [Online] Available at: http://www.twitter.com [Accessed 04 June 2012].

[20] Crockford, D., 2006. *The application/json Media Type for JavaScript Object Notation (JSON)*. [Online] JSON.org Available at: http://www.ietf.org/rfc/rfc4627.txt [Accessed 30 May 2012].

[21] Fielding, R., 2000. *Architectural Styles and the Design of Network-based Software Architectures*. [Online] Available at: http://www.ics.uci.edu/~fielding/pubs/dissertation/top.htm [Accessed 05 June 2012].

[22] Rodriguez, A., 2008. *RESTful Web services: The basics*. [Online] Available at: https://www.ibm.com/developerworks/webservices/library/ws-restful/ [Accessed 02 August 2012].

[23] W3C, 2000. *Simple Object Access Protocol (SOAP) 1.1*. [Online] Available at: http://www.w3.org/TR/2000/NOTE-SOAP-20000508/ [Accessed 06 May 2012].

[24] Wikipedia, 2012. *Object-relational mapping*. [Online] Available at: http://en.wikipedia.org/wiki/Object-relational_mapping [Accessed 03 June 2012].

[25] Django, 2012. *The Web framework for perfectionists with deadlines*. [Online] Available at: https://www.djangoproject.com [Accessed 27 May 2012].

[26] Flask, 2012. *Flask, Web development one drop at a time*. [Online] Available at: http://flask.pocoo.org/docs/ [Accessed 28 May 2012].

[27] Joyent, 2012. *Node.js*. [Online] Available at: http://nodejs.org [Accessed 28 May 2012].

[28] Rails, 2012. *Ruby on Rails*. [Online] Available at: http://rubyonrails.org [Accessed 28 May 2012].

[29] Apple, 2011. *What's New in Xcode 4*. [Online] Available at: https://developer.apple.com/technologies/tools/whats-new.html#llvm-compiler [Accessed 17 June 2012].

[30] Xamarin, 2012. *MonoTouch*. [Online] Available at: http://xamarin.com/monotouch/ [Accessed 28 May 2012].

[31] Apple, 2012. *iOS Developer Program License Agreement*. [Online] Available at: https://developer.apple.com/programs/terms/ios/standard/ios_program_standard_agreement_20120611.pdf [Accessed 30 May 2012]. Requires an Apple developer account to view.

[32] Bookwalter, J.R., 2010. *Apple Facing Federal Probes Over Section 3.3.1 of iPhone SDK?* [Online] Available at: http://www.maclife.com/article/news/apple_facing_federal_probes_over_section_331_iphone_sdk [Accessed 29 May 2012].

[33] Apple, 2012. *iOS 6 Preview*. [Online] Available at: http://www.apple.com/ios/ios6/ [Accessed 19 May 2012].

[34] Apple, 2011. *What's New in Xcode 4*. [Online] Available at: https://developer.apple.com/technologies/tools/whats-new.html#llvm-compiler [Accessed 17 June 2012].

[35] Siracusa, J., 2011. *Mac OS X 10.7 Lion: the Ars Technica review*. [Online] Available at: http://arstechnica.com/apple/2011/07/mac-os-x-10-7/10/ [Accessed 17 June 2012].

[36] Siracusa, J., 2011. *Mac OS X 10.7 Lion: the Ars Technica review*. [Online] Available at: http://arstechnica.com/apple/2011/07/mac-os-x-10-7/10/ [Accessed 17 June 2012].

[37] Apple, 2011. *What's New in Xcode 4*. [Online] Available at: https://developer.apple.com/technologies/tools/whats-new.html#llvm-compiler [Accessed 17 June 2012].

[38] Dropbox, 2012. *Dropbox*. [Online] Available at: https://www.dropbox.com [Accessed 20 May 2012].

[39] Git, 2012. *Git Version Control*. [Online] Available at: http://git-scm.com [Accessed 01 June 2012].

[40] Coia, G., 2012. *github*. [Online] Available at: https://github.com/giobox [Accessed 20 May 2012].

[41] Bare Bones Software, 2012. *BBEdit 10*. [Online] Available at: http://www.barebones.com/products/bbedit/ [Accessed 20 May 2012].

[42] Omni Group, 2012. *OmniFocus for Mac*. [Online] Available at: http://www.omnigroup.com/products/omnifocus/ [Accessed 20 May 2012].

4. Implementation & Testing

This chapter outlines the implementation and testing process used in the development of the system. The first section of this chapter explains the implementation of the website and its REST API. The following section describes the iOS application. At the end of this chapter there is some discussion of testing approaches and methodologies that were adopted.

4.1 iACT Web Application

Having taken the decision to use Ruby on Rails to implement the site during the design phase I then moved forward to development. Given I had no experience in Rails development work I consulted a number of textbooks on the best way to proceed. Following the excellent book on Rails development by Michael Hartl[43], I was able to learn the framework and implement the site simultaneously. Ruby on Rails adheres to a strict model view controller architecture style. An outline of the class structure of the web site can be found in the documentation following this chapter. Typically each of the site's views corresponds to a main page template on the site. The view's controller, pulling relevant records from the database, then renders the view with appropriate data. Rails helpfully abstracts all interactions with the database so that the developer writes no SQL commands. Instead one can interact with records in the database as if they are native Ruby objects. Behind the scenes Rails takes care of performing the relevant SQL and converting this to and from the native Ruby. The major advantage of this is that one can change the database backend technology completely without changing a single line of the model code. Telling Rails what database software you are using is enough for it to automatically switch to whichever commands are needed to carry on working with the new database software. This makes it incredibly easy to scale to bigger more powerful database back ends should the site grow substantially.

Hartl's book describes an excellent methodology for the design and creation of Ruby on Rails web sites. Following his advice, I put the site under git source control and used a test-driven-development process. With this process, one writes the test for the site functionality before any implementation takes place. Naturally, these will fail the first time the suite of tests is run. One then works until each of the failing tests passes, at which point there should be a correct implementation of the required functionality. This also leaves the developer with a full suite of tests that can be reused as the site grows to ensure later functionality does not break existing components of the web site. This is done using a rails add-on library called *RSpec*[44].

The ability to quickly add functionality such as RSpec to the site was one of the key advantages of using a WAF. Rails allows the developer to quickly install ready built software components called Gems. These are installed simply by specifying the required Gem in the gem file in the application's root folder. One then simply types 'rails bundle install' in a terminal window and the packages listed in the gem file are automatically installed.

4.1.1 The Web Site Model

One of the first steps I took in creating the web site was to decide exactly what would be modeled and how this would be displayed. In my original design documents I had outlined a database schema along with some wireframes for the basic views. Using this as a starting point, I was able to correspond these wireframes to appropriate views and view controllers in Rails. I then created appropriate model files from which Rails would automatically generate the database. By default Rails is setup to use SQLite 3 for the database back end. I saw no real reason to change this during the development of the site. The model adopted for the website is shown in *figure 12*. Both the website and the iOS application would end up with databases that are very similar, but with some key differences.

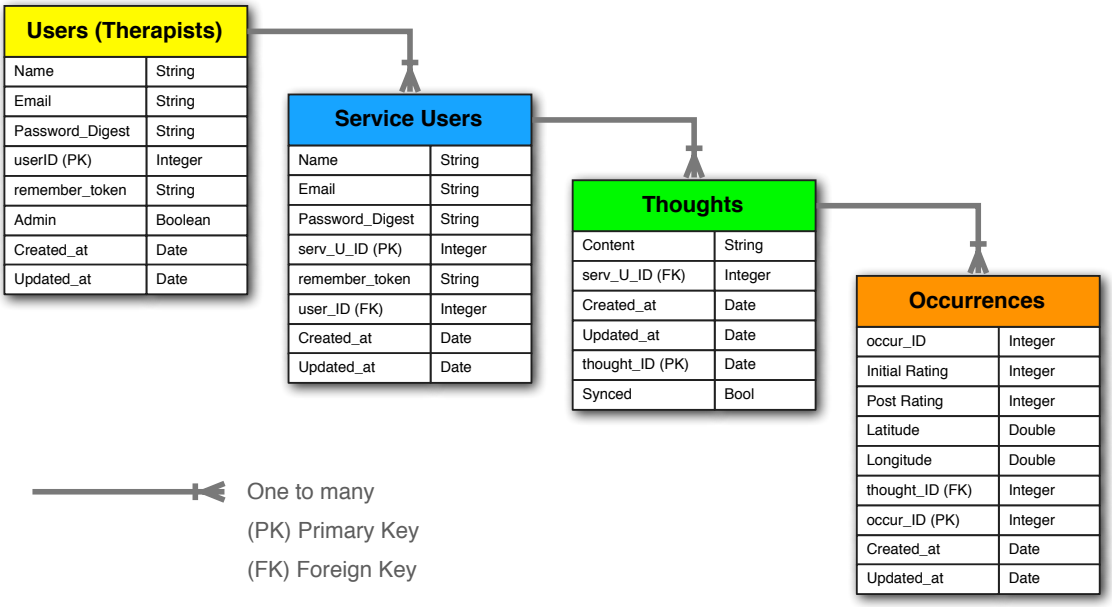


Figure 12 Website Data Model

The iACT website has one additional class of entities to model in comparison to the iPhone application, and that is the therapist users. There are two types of therapist users: administrative and non-administrative. This is set in the database with the admin Boolean attribute. Both types of user are identical in every way, except that the administrative user is granted the ability to add and remove other therapists from the system in addition to adding service user accounts. A non-administrative therapist may only add service users. All relationships are one to many. This means each therapist can have many service users, who in turn have many thoughts with many occurrences.

Another concern in the design of the database was syncing thoughts back to the user's iPhone. As the web interface permits the therapist to add thoughts to be synced back to the user's handset, I wanted some way to ensure only new thoughts are sent. This is because the iPhone application is a mobile application, and may be used in situations where there is poor connectivity over a mobile data connection, or the user might be billed according to data usage. Keeping the data sent and received to a minimum was therefore a priority. To ensure that only thoughts that have not been synced are sent to the client's handset, an extra Boolean, 'synced', is present on each thought record. When a therapist adds a thought through the web interface this is set to false. When the service user performs a sync operation through the REST API only those thoughts

whose sync boolean is false are sent. Once they have been sent the status of the thought is changed to true, and this thought will not be sent again. This means that if the user refreshes the thought list on their iPhone and there are no new thoughts, no data is sent.

In designing the database I have also tried to ensure that database is compliant with the Data Protection Act[45]. All of the information gathered is absolutely necessary to the functioning of the site. There is no spurious or unnecessary information being collected. In addition, by storing the thought and thought occurrences in separate tables from the service users themselves, it is possible to analyse the recorded thought data anonymously. This means that any research on the raw thought data will only contain the user's ID; there will be no identifying information. The REST API also allows the service user to easily remove thoughts from the server, ensuring that only data they wish to be recorded is actually saved.

4.1.2 The Web Application User Interface

When I first set out to design the site in my proposal I had outlined some detailed wireframes. I initially chose to implement these using HTML5 and CSS3 to create template pages for the web site. Some of these prototype pages can be seen in *figure 13*. Shortly after creating these prototypes I gained yet more valuable advice from Michael Hartl's Rails development book. Rather than create all of my visual elements from scratch, I discovered an extremely useful open source tool from Twitter called *Bootstrap*[46].

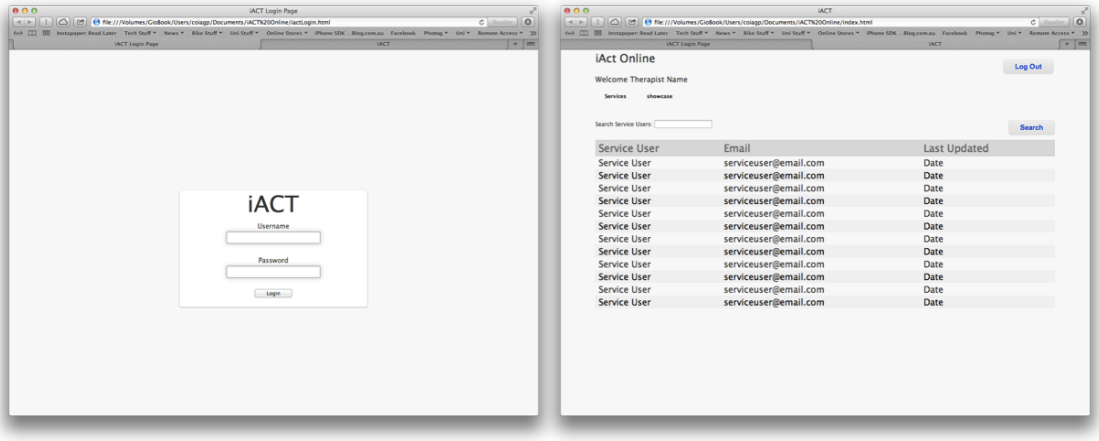


Figure 13 Original iACT Site Interface Prototype

Bootstrap is a set of free tools and templates for many of the common web user interface scaffolding elements such as page layouts, forms, buttons, text, headings and so forth. Using these tools, one can rapidly build complex and well-designed user interfaces with an extremely high quality feel. Bootstrap however uses the LESS-CSS language to create its dynamic style sheets, which is not supported by Ruby on Rails. This was easily remedied once again by specifying an extra library in the gem file that converts the LESS-CSS to a format understood by Rails. LESS-CSS allows the creation of much more powerful style sheets by adding support for variables and functions. At runtime these sheets are then dynamically translated into the conventional CSS for rendering in a browser.

Using the provided Bootstrap user interface elements is as simple as specifying the appropriate class tags in the page's HTML code. One can either use Bootstrap's defaults or over-ride them through the use of custom CSS. Given the short time available to develop the site I made very few changes to the defaults. The standard Bootstrap elements have such a high quality feel to them that I did not think it a valuable use of my limited development time to customise these to any great extent. Despite this, the Bootstrap based user interface has ultimately managed to remain remarkably close to the original design wireframes. Some examples of the Bootstrap based interface can be seen in *figure 14*.

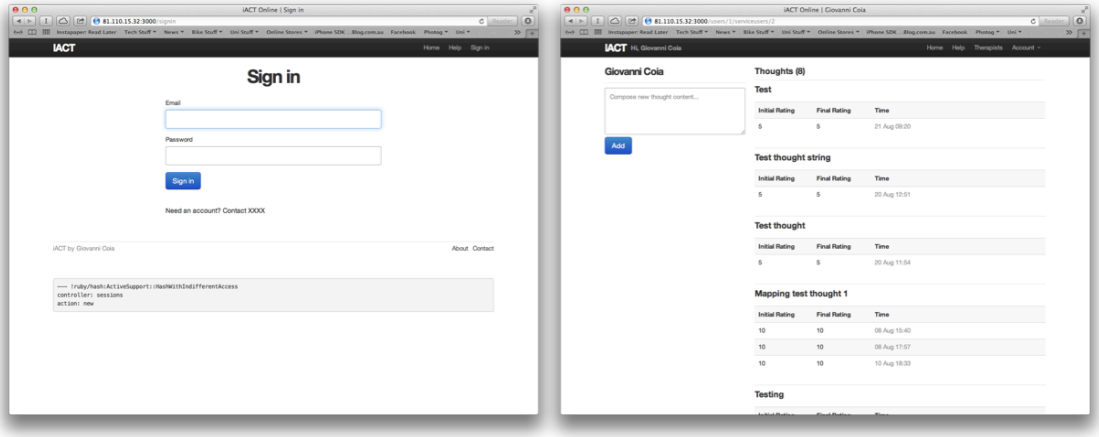


Figure 14 iACT Bootstrap Based Interface

Another key consideration in the design of the user interface was to ensure that data was presented in an easy to understand form. As there is the potential for a user to record a great number of thoughts on the system, ensuring the display of results did not overwhelm the user was of paramount importance. To improve this I decided to adopt a paginated display of results. It is a standard convention for a web site to spread a large number of listed items across multiple pages with a page selector control underneath, but the code to do this can be quite complex. Fortunately I did not need to implement any of this myself. Pagination support was usefully provided by the will_paginate library[47], which, like the other libraries I used, was added simply by specifying it in the gem file. Pagination then became as easy as adding one line to the view:

```
<%= will_paginate @posts %>
```

This line takes the array of posts, generated by the view's controller, and automatically creates the pagination links. Even better, these pagination links are then automatically styled to look nice by Bootstrap as demonstrated in *figure 15*.

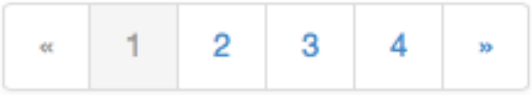


Figure 15 Bootstrap Pagination Element

4.1.3 Securing the Web Site

With the model and the interface to access its data built I then considered the security requirements of the site. As it is intended to store and display sensitive information about the iPhone application's users it was critical that access to the web site be secure. To facilitate securing the web site a number of approaches were adopted.

In order to allow user login functionality, one of the main changes made to the original proposal for the database design was to store a password digest rather than the password itself. This was done using an extra package added to the WAF called Bcrypt[48]. Using Bcrypt to store an encrypted hash of the password allowed me to validate the therapist and service user's passwords without actually storing the password itself. This affords the site an additional layer of security in that should the database be accessed maliciously the attacker will still be unable to log into the site.

A second change made to the database was to add a 'remember token' to each user and service user. Because HTTP is a stateless protocol there is no way to easily maintain a constant connection with a logged in user. The approach I adopted to stop the user having to log in with every single request was to randomly generate a long string of text every time a user is added to the database. This is saved in the user's remember_token field. When the user signs into the site a cookie is placed on the user's computer containing this token. This can then be used on subsequent requests to confirm that the user is logged in. This is a basic but effective approach to managing sign-in sessions, and this is why the login controller is referred to as the 'sessions controller' in the class documentation.

For securing a website there is a huge range of techniques and software packages available. I initially considered adopting OAuth[49], a well-known open source solution for securing HTTP services. However, given the limited development time, the Bcrypt password hash and remember token based solution was simple to learn and easy to implement. Any view controller that requires an authenticated user simply has the following line added:

```
before_filter :signed_in_user
```

This line of code ensures the controller checks for the presence of a valid remember token cookie on the user's browser, redirecting them to the sign in page if it is not. The code above calls a method specified in the sessions_helper.rb file that contains the authentication code. A further test can be applied to ensure that not only is the user signed in, but also that it is a user who has permission to view the content. This ensures that therapists can only view their own service user's data, and not the data of other users:

```
before_filter :correct_user,   only: [:edit, :update, :show]
```

The above code, lifted from the thoughts controller, ensures that the edit, update and show methods in the controller can only be called by a user who is permitted to view it. Again, a redirect will take place in the event of an inappropriate access attempt.

One of the drawbacks of the token-based authentication system is that it is potentially insecure when used over an unsecured HTTP connection. There have been demonstrations of attacks where the token has been intercepted[50], particularly over wireless Internet connections. Were a user to intercept the token this can be inserted into a cookie on their own computer, allowing them to access restricted site content as if they are logged in without having to enter any username or passwords. This technique has grown much simpler in recent years, with Firefox browser plugins available to do the work automatically for a malicious user[51]. The standard solution to this problem is to ensure all communications take place over HTTPS, and this is a consideration that must be taken into account when setting up a Rails server with my code.

This security implementation provides an appropriate level of security for browser access. However, given my API for the iPhone client uses REST it too is HTTP based. A similar technique can be therefore be used to secure REST access as well. The iOS application can download the remember token when the user logs in and then submit it with each REST request.

4.1.4 The REST API

Once the web site was fully functional I set about implementing the REST API to allow the iPhone clients to interact with it. This was handled by means of an extra controller, known as the iPhone sessions controller. This controller performs two tasks, handling both authentication of the iPhone clients as well as the synchronisation of thought data. Typically most controllers in a Rails web application will have a view that formats the output into some kind of HTML for displaying in a client browser. However, this controller does not need to provide content for a browser. Unlike all the other controllers in the Rails application it has no view. Instead the iPhone session controller returns either HTTP status codes or JSON.

Implementing the API proved one of the biggest challenges of the development of the site. My eventual solution relies on the same principles as REST, but its implementation is not technically fully compliant with the definition of a RESTful API. A fully formed REST API utilises the HTTP GET, POST, PUT and DELETE requests for specific tasks. The API I have developed can perform all of the relevant CRUD (Create, Read, Update and Delete) tasks, but rather than using specific HTTP request types for specific operations all of the requests take place as HTTP GET requests. Depending on the operation required, a different URL path is specified. With this path an HTTP GET request is sent with some parameters in the HTTP header. These parameters are interpreted by the iPhone sessions controller and the appropriate CRUD action takes place. The table in *figure 16* outlines precisely how the API operates.

Action	URL	Parameters	CRUD Action	Successful Response	Unsuccessful Response
iPhone Login	/iphonelogin	Email, Password	CREATE	JSON containing User data	HTTP Response 403
Download new Thoughts	/getthoughts	User ID	READ	JSON containing non-synced thoughts	Empty JSON
Delete Thought	/deletethought	User ID, Thought Content	DELETE	HTTP Response 200	HTTP Response 404
Add Thought	/recordthought	User ID, Thought Content	CREATE	JSON containing the newly saved thought	HTTP Response 403
Add Occurrence	/recordoccurrence	User ID, Thought Content, Initial Rating, Post Rating, Latitude, Longitude	UPDATE	JSON containing the newly updated thought	HTTP Response 403

Figure 16 REST API Specification

As shown above, an HTTP GET request can be sent to the relevant URL. These URLs are reserved solely for the REST controller, and are not used to return any HTML when accessed by a web browser. The HTTP GET request must contain the relevant parameters in the HTTP header. If the request is a success, a response in the format shown will take place, otherwise a failing response is sent. Knowing precisely how the above API operates means one can construct another application or website that uses the API to obtain and transmit content. Having both a successful response and a failing response makes it easy to validate responses in the requesting application. This API can therefore be used with iACT client applications on other platforms in the future.

It should be noted that while I have implemented much of the code for authenticating the iPhone clients, I was not able to get this code fully functional in time for submission. This means that although browser based

access is presently secured the REST API is exposed. Whilst the API URLs present no browser-based interface to interact with, a malicious user could in theory insert records or delete data from the database if they construct the HTTP GET requests with the correct parameters. This would of course require rectification before using iACT in a real production environment.

4.2 iOS Application

After having developed a working web application for iACT online with a functioning and documented REST API, I was now able to begin implementation of the iOS application. This work would take place within the Xcode IDE and proved to be some of the most challenging and enjoyable tasks in the project. In order to learn iOS development I worked through two textbooks. The first was *Sam's Teach Yourself iOS 5 Application Development in 24 Hours*[52] which gave me a good grounding in the basics of iOS development. This was then followed by some work with *Beginning iOS 5 Development: Exploring the iOS SDK*[53] to gain an understanding of the more complex systems in iOS. Finally I followed a great deal of Stanford University's online iOS development course[54].

4.2.1 Working with Xcode

In the main, iOS application development in Xcode is a largely positive experience. One can quickly sketch out user interfaces then graphically link these to methods in the code simply by clicking and dragging. As a result, there is often very little coding involved in user interface creation, leaving the programmer free to focus on application logic. Xcode also forces the developer to adhere to a strict model view controller architecture. This is particularly helpful, as entirely new user interfaces can be created without having to edit the controller or the model. This also makes it very easy to use the same code in desktop or iPad versions of your application should one wish to port to either or these platforms, with in many cases only a new view required. This is in large due to the iPad sharing the same OS and the Mac OS sharing many of the same core libraries and frameworks.

That said, Xcode is by no means a perfect tool and there where some frustrating aspects I encountered relative to other IDEs I have used. The biggest, and frankly strangest of these, is the lack of any function to automatically create method stubs for interfaces (or protocols, to use the Cocoa/objective-C parlance) you implement in the code. This omission is stranger still when one observes that Xcode checks your code to make sure you have implemented all the required methods. Xcode therefore is clearly aware of the required methods, and as you start to type them will even autocomplete the method signature if you have declared the interface in the class declaration. If Xcode is already aware of this, it seems like automatic completion for method stubs would only be a matter of a few lines of code at most. I was initially convinced that I must have been missing the option, but research on the matter online revealed this to be one of the most common complaints regarding working with Xcode.

When developing iACT I quickly grew tired of constantly having to check the documentation then cut and paste the interface's required methods and searched online for a solution to this problem. I quickly came across a utility called "Accessorizer"[55] which added much of the missing automatic code generation features one finds in other popular IDEs such as Eclipse or Visual Studio. Xcode is in a constant state of development and its feature set has evolved enormously. One can only assume it is a matter of time before this shortcoming is addressed.

4.2.2 Coding with Storyboards

As discussed in the previous chapter, the introduction of the Storyboards feature in iOS 5 has seen the adoption of a new default design pattern for most iOS applications. In the main, the iACT application consists of a series of view controllers linked together by means of a navigation toolbar known as an NSNavigationBar. Where appropriate, these view controllers communicate with a singleton class containing the model for the application. If needed, the view controllers also sometimes talk to other view controllers to pass on

information. The views of course never talk to the model. The model in this case stores all the user's recorded thoughts. The use of an NSNavigationBar results in a fairly standard user interface design for an iOS application with an interface paradigm that is found throughout the iOS, meaning most users should intuit how to operate the software almost immediately.

When creating a storyboard-based application, one graphically arranges the views that will form the interface for the application as shown in *figure 17*. One then joins these views together with transitions known as 'segues'. As one moves through the application from one view to another, the appropriate segue is called in the code. A diagram providing an overview of the application's classes and their interactions with each other can be found in the documentation at the end of this chapter.

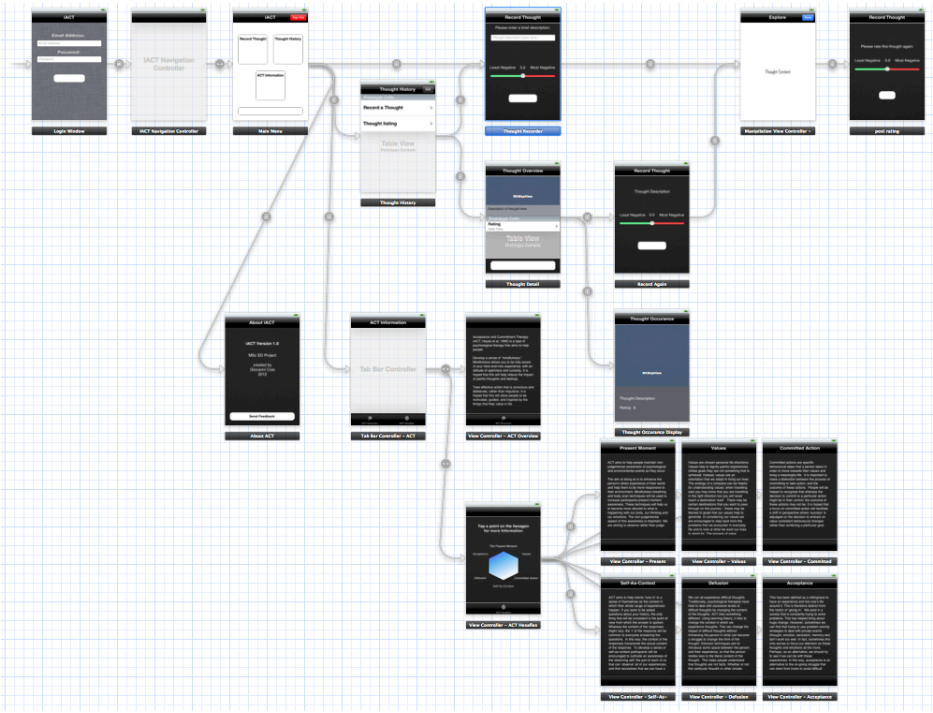


Figure 17 iACT Storyboard

Following the creation of the application's storyboard, it was then necessary to implement a model to provide data for the application's views.

4.3 iACT Application Data Model

The data model present in the iPhone application formed one of the most important aspects of the system. As one of the primary goals of the application was to provide a means of recording emotional events it was critical that these be stored in a permanent and easily to manipulate fashion. The application model was also one of the elements of the implementation that saw the greatest degree of change as the project progressed.

4.3.1 Model Overview

The data structure for the iACT mobile application consists of three entities and their attributes. Each entity has a one-to-many relationship with its child entities. The result is that *users* have many *thoughts* that in turn can have many *occurrences*. The structure of the data model is outlined in *figure 18*.

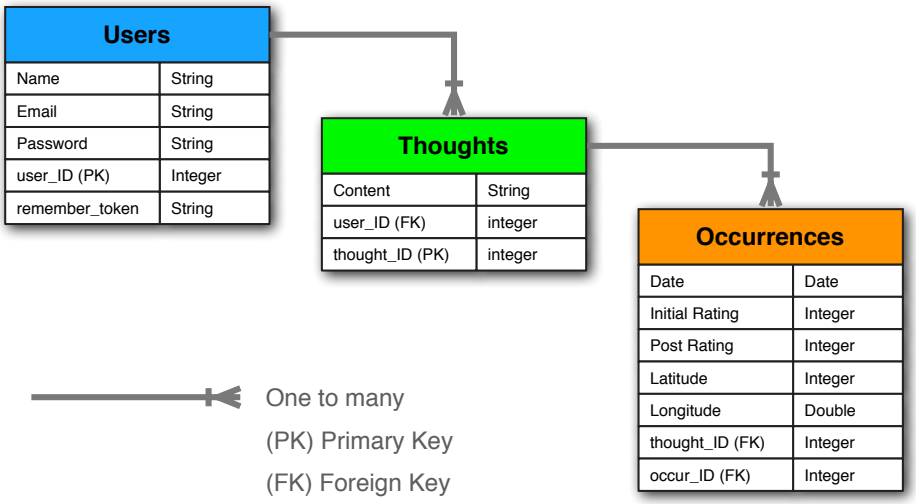


Figure 18 iOS Application Data Model

Each thought recorded by the user contains a description of that thought stored as a string in the content attribute. Every time a thought is recorded an occurrence of that thought is logged. This structure permits the same thought to be recorded multiple times simply by adding a new occurrence. This design was chosen because it is likely that users of the application will want to be able to record recurring thoughts. Each occurrence logs the date and location it was recorded, along with two ratings. The first rating is recorded before the user has interacted with the thought in iACT, the second afterwards. It is hoped that this will in time allow the application to measure a reduction in the negative feelings associated with a thought after having used the software.

4.3.2 Object Archiving

Initially my first attempt at creating a data model saw a very traditional approach taken. I wrote a singleton class called the dataModel that contained an array of thoughts, and each thought contained an array of thought occurrences. This dataModel was then instantiated by the application delegate class at launch. This could then be saved or loaded by an archiving process. Archiving in Cocoa Touch is in practice a form of object serialization[56], a technique common to other programming languages such as Java.

Object archiving allows the writing of an instance of a class to disk in order to save its state. The object can then be restored from this state and a new instance created from it. In order for this to work the class one intends to archive must conform to the NSCoding protocol. To conform to this protocol, the class must contain methods to encode and decode the instance variables to disk. Each instance variable of the object is then stored against a key in a property list file. When rebuilding the instance with the decoder method the appropriate keys are used to get the data back into the instance variables.

I used this NSCoding based approach in the earliest versions of the application. When a user logged in for the first time, a file was written to disk with the user's name appended to the end. This ensured that if multiple users used the same phone they would not overwrite each other's data. When the user pressed the home button to return to the home screen of their iPhone a save data method in the app delegate would be called, writing the model to the disk. Thereafter each time the application was launched again the data would be decoded from the archive and the model arrays populated.

This approach works for very small-scale applications, but quickly issues begin to arise. Firstly, this style of model is grossly inefficient, and is not recommended for data structures over one or two megabytes in size. Although the data set is unlikely to grow to extreme sizes where efficiency might become problematic it was clear there was almost certainly better ways of storing this data. Modeling one to many relationships required

nested arrays. This meant that retrieving nested attributes was computationally very intensive as the parent array was searched, then its child was searched, then its child too was searched and so forth. Secondly, adding extra data fields to save to the model was quite labour intensive, requiring careful creation of new getter and setter methods. It was also extremely difficult to validate the contents of the model, as one could not simply examine the contents of the file as one might a database. Finally if the application crashes or quits unexpectedly, there is a real chance that there will be some unsaved data that will be lost.

4.3.3 Data Model Alternatives

As the list of things I wished to store grew debugging the object archive model quickly become a major issue and was taking up far too much of the limited development time. I decided late on in the project to examine alternative structures for the model. With the iOS frameworks there are in essence three other options for saving and persisting data.

The first and most basic is of course a simple text file. Data could be turned into a comma separated value (CSV) format and this could then be printed into a text file. This is an even poorer solution than the object archiving method I had been using until now and was quickly discounted. Having to read and write from the file would introduce enormous scope for string formatting errors resulting in corrupt data or crashes and would clearly become problematic to deal with. One could improve matters slightly by using XML or JSON for the output file, but these would still require complex parsing methods to return native objective-C objects.

The second option was to use a database. All iPhones with iOS 5 run an embedded version of SQLite 3, the popular database software. SQLite is advantageous in that it would bring some big improvements in the efficiency of the data model. I would also be able to remove much of the file writing code. However, the code for interacting with SQLite is very complex. The developer has to write the native SQL commands into the application and very quickly the code becomes messy and confusing. It is necessary to continually pass strings containing the SQL to the database, rather than rely upon normal objective-C message calls.

The final option was the most complex with regards to initial setup, but had clear longer-term benefits. With the introduction of iOS 3, Apple introduced a new framework to iOS called Core Data. Core Data is an object relational mapping system that works in a similar manner to the way in which Rails abstracts the SQL from the web site model's implementation code. As with Rails, instead of having to deal with SQL queries, you simply work with native objects and in the background Core Data takes care of the SQL for you. This means new database records can be created simply by instantiating a class that is linked to that record in the database. Attributes can be set as instance variables of that object, rather than editing the database directly. Similarly, searches simply return arrays of instantiated objects corresponding to results, rather than raw SQLite table data. Perhaps the best feature of Core Data is shown in *figure 19*. One can create and model the database structure through a simple graphical interface, the code for the corresponding object classes and SQL table creation is then generated automatically by Xcode.

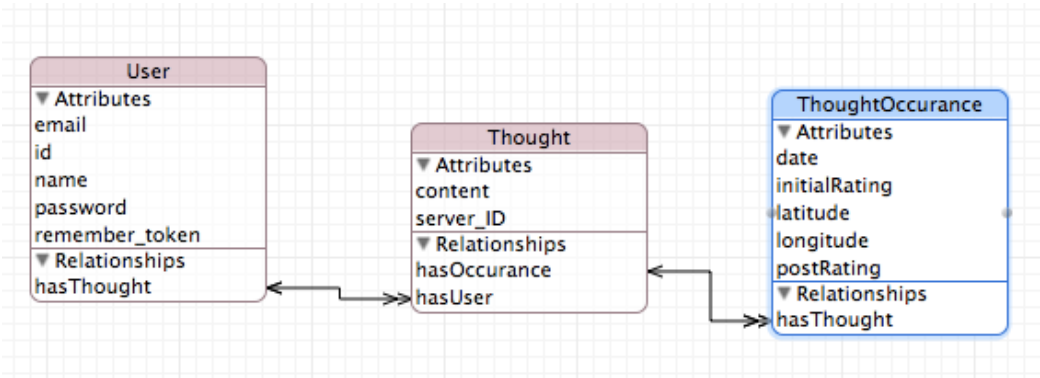


Figure 19 Core Data Model Creation

Because the corresponding object code is automatically generated, this makes adding extra complexity to the data model very easy. If it is decided at a later date to record some extra kind of data it can be added to the model almost instantly.

All of this powerful functionality however does not come for free. There is a significant learning curve to using Core Data and there is also a large amount of 'boiler-plate' code to add and maintain in order to use the framework. Accordingly it took some time to get the Core Data model to work. However, with it now working, it is a solid platform on which to add functionality to in the future and has resulted in significantly more efficient and robust code than the original model. It also opens the door to more advanced iOS features such as iCloud support at a later date.

4.4 Syncing with iACT Online

With the model now functioning, the next task was to consider how to build the code to sync using the web site's REST API. This arguably presented the greatest challenge in the implementation of this project. The iOS application had to perform three primary syncing tasks. Firstly, it had to be able to submit a newly recorded thought to the server so that the therapist could check the recorded data. Secondly, it had to be able to download any new thoughts from the server and add these to the local data model so that the user can record some new occurrences of that thought. Finally, the application needed to support login detail validation.

Initially I considered writing my own code to facilitate communication with the web site. iOS provides a built in class called NSURLConnection to facilitate HTTP connections. The intent was to send the appropriate REST calls to the website API then handle the returned information using this class. When talking to the web site there is in essence three outcomes to handle. The first is the simplest: no connectivity. The second is that some JSON is returned, and the third is that an HTTP status code is returned. The complexity of this problem quickly grew. As I began constructing the synchronisation code it was apparent that requests would need to be multi-threaded, for otherwise a slow data connection might lock up the user interface. Similarly, a slow connection might cause the thought recording requests to start queuing up and I did not want to lose any data. Given the popularity of the REST API architecture I quickly assumed that there was very likely to be a public open source library for handling this kind of problem. Given the large scope for bugs I was keen to find a tried and tested code base if possible.

After a small amount of research I quickly came across RestKit[57]. RestKit is a freely available open source library for use with iOS applications that provides REST API call functionality. It is already multi-threaded and supports queuing requests. It also provides an object mapping system to automatically map the returned JSON against native local objects. This means that when used with Core Data, it can in theory automatically insert new records into the database.

Sadly I was not able to implement RestKit to its fullest advantage. At present I am using it only to submit thoughts and handle the receiving of the JSON files from the server. RestKit also handily deals with HTTP response codes and I was able to quickly set up some conditional behaviors according to the server response. However, I was not able to get the native object mapping to work. RestKit's development is unfortunately several steps ahead of its documentation, meaning it was often quite difficult to establish just how to use it at times. At present the documentation for the object mapping is just too out of date and I did not have enough time to examine the complex code thoroughly enough to gain an understanding.

That said, when combined with iOS 5's built in JSON parsing classes it still made for a powerful syncing system. Rather than use RestKit's object mapper I instead used iOS's NSJSONSerialization class to convert the JSON to a dictionary of objects and keys. One can then loop through all the entries in the dictionary and allocate new objects for each, establishing instance variables from the relevant keys. Coupled with Core Data, this performs much the same function as RestKit's object mapping would have done.

4.5 The View Controllers

As the iOS application used the new storyboard feature for its design, its structure is largely centered on its view controllers. Following the creation of a working data model and synchronization system I was able to focus on the user interface elements. This section provides detail on the implementation of these views and their controllers.

4.5.1 Login View

Upon first launching the application the user is presented with the login screen view shown in *figure 20*. This view performs several functions. Firstly, it checks if the user has already logged in. If this is the case, it automatically transitions to the main menu. This means the user can stay logged in until they decide to log out, removing the frustration of logging in each time the user launches the application. For the storage of data such as login credentials and usernames it is not actually necessary to create your own data structure. For small but important pieces of data like this every iOS application maintains a property list called standardUserDefaults. This can be accessed by instantiation of the singleton class NSUserDefaults.

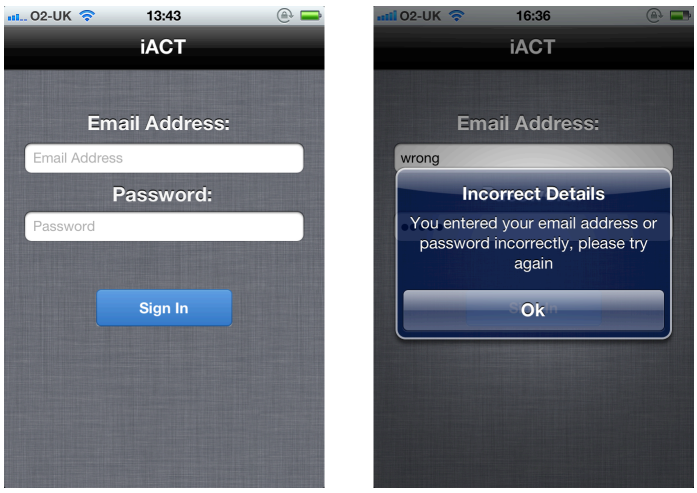


Figure 20 iACT Login Screen View and Failed Login Message

When interacting with NSUserDefaults one can store small pieces of data in the property list by providing the data along with a key to retrieve it. For example, following a successful login, iACT stores the following in standardUserDefaults:

```
[userDefaults setObject:emailAddressField.text forKey:@"username"];
[userDefaults setObject:passwordField.text forKey:@"password"];
[userDefaults setBool:YES forKey:@"loginStatus"];
```

Figure 21 NSUserDefaults Example

This code takes the content of the email address and password fields and stores them in the property list against the keys ‘username’ and ‘password’. Similarly, a Boolean value YES is stored for the key ‘loginStatus’ to indicate that the user successfully logged in. This value is checked each time the application launches to determine whether or not the user has to login or has saved login details. When the user ultimately decides to log out this will of course be set to NO and the values of username and password cleared.

Retrieving data from standardUserDefaults is equally easy. One just specifies the key of the object you wish to retrieve:

```
[userDefaults boolForKey:@"loginStatus"];
```

The above retrieves the boolean value for loginStatus. For many simple applications NSUserDefaults is all the model code that will be needed. NSUserDefaults can easily be called from almost any object class in iOS.

As one would expect, the login process only succeeds if the user enters valid credentials. iACT only allows users who have had an account created by a therapist on the iACT web site to log in. Validation takes place using the RestKit framework. A REST login request is constructed using the user's email address and password. The iPhone sessions controller on the iACT website then performs the necessary validation. If the credentials are correct, JSON containing user data is returned. From this the user's full name will be stored for display on the main menu view and then a transition to the main menu will take place. Should the user's credentials fail to validate an error will be displayed, after which the user will be returned to the login screen.

4.5.2 Main Application Layout

Once a user has successfully logged in, the main menu is displayed. This will now be the first view the user sees upon application launch until such time as they tap the logout button. From this point forward, the view is entirely built around an NSNavigationBar. NSNavigationBars are among the most common user interface elements one finds in iOS and provide the bar at the top of the screen with titles and navigation controls as shown in *figure 22*. Using this bar, one can navigate linearly backwards and forwards through a series of views. NSNavigationBars work by maintaining a stack of views. Each time the user navigates deeper into the view hierarchy the new view is 'pushed' onto the NSNavigationBar view stack. Similarly, when the user navigates back, the top view is 'popped', revealing the view underneath. Typically the NSNavigationBar will always show a back button to the previous view on its left side with the name of the previous view written on it. Eventually the user will return to the top parent view if they press this enough times. Moving through the view stack causes a sliding animation to the next view, giving a strong visual cue the user is moving through a series of linked screens.

Figure 22 NSNavigationBar

Each of the views linked by the NSNavigationBar typically has its own view controller. In many instances this does not actually need to interact with the data model, but instead collect some input for use in the next view. One of the most important aspects in using the NSNavigationBar is therefore ensuring that a parent view can pass information onto its child view in the stack, or vice versa. For example, in iACT, the user must rate the same thought twice when passing through three different views. Thoughts are modeled as thought objects in the application; I therefore wanted to store all the data about that thought in the same thought object. It was necessary then to pass the thought instance through the various views as the user interacts with it.

This is an extremely common design problem in iOS applications and there is some default methods provided to deal with this issue. *Figure 23* illustrates the standard approach using the prepareForSegue method. By default, when transitioning to a new view, iOS always checks first to see if there is a prepareForSegue method present in the view's controller. If there is it is called, and this method is run before the new view is loaded. Knowing this, you can use this method to setup the next view so it is ready when it appears.

```
-(void)prepareForSegue:(UIStoryboardSegue *)segue sender:(id)sender {
    if ([[ segue identifier] isEqualToString:@"manipulateThought"]) {
        //pass the created thought and occurance to the next view
        ACTManipilationViewController *manipulationViewController = [segue
destinationViewController];
        manipulationViewController.thought = self.thought;
        manipulationViewController.occurance = self.occurance;
        manipulationViewController.newThought = self.newThought;
    }
}
```

Figure 23 prepareForSegue method

The code above provides an example of the method's use. When laying out a storyboard, each transition or segue should be given a name to identify it. This identifier can then be checked to find out exactly which segue is taking place. This code therefore confirms that if the segue is called 'manipulateThought', the code should instantiate the next view's view controller, 'manipulationViewController', and set some of its instance variables using data from the currently displayed view. In this case the thought presently being recorded is set as a property of the new view so it can carry on working with it after the transition completes.

It is not strictly necessary to test the name of the segue, but one must remember that with NSNavigationBar the user can segue in two directions, backwards and forwards. In this instance I only wanted the data passed in the event the user moves forward through the view hierarchy, so testing for the segue identifier ensures the thought object is only passed when necessary. Of course I will eventually want to save changes back to the data model and sync with the web site, but the prepareForSegue method is the primary means by which view controllers communicate with one another.

The prepareForSegue method is not the only solution to inter-controller object communication. Before discovering the prepareForSegue technique I had put some instance variables inside the app delegate singleton class to use as a sort of temporary storage space for the controllers. The parent view controller would set an instance variable in the app delegate to point to the object I wanted to pass, the child view could then get this from the app delegate. However, using the prepareForSegue technique results in cleaner code and less coupling between classes. Rather than a three-way relationship between the parent, app delegate and child classes, there is now a direct relationship between the two communicating controllers. Another solution used in some of the iOS development textbooks is to put some instance variables inside the NSNavigationBar's controller. As the NSNavigationBar is passed to each view in the navigation bar stack its instance variables are accessible by all of its views. This still results in the same drawbacks as using the app delegate however, so I chose not to adopt this approach.

4.5.3 Recording a Thought View

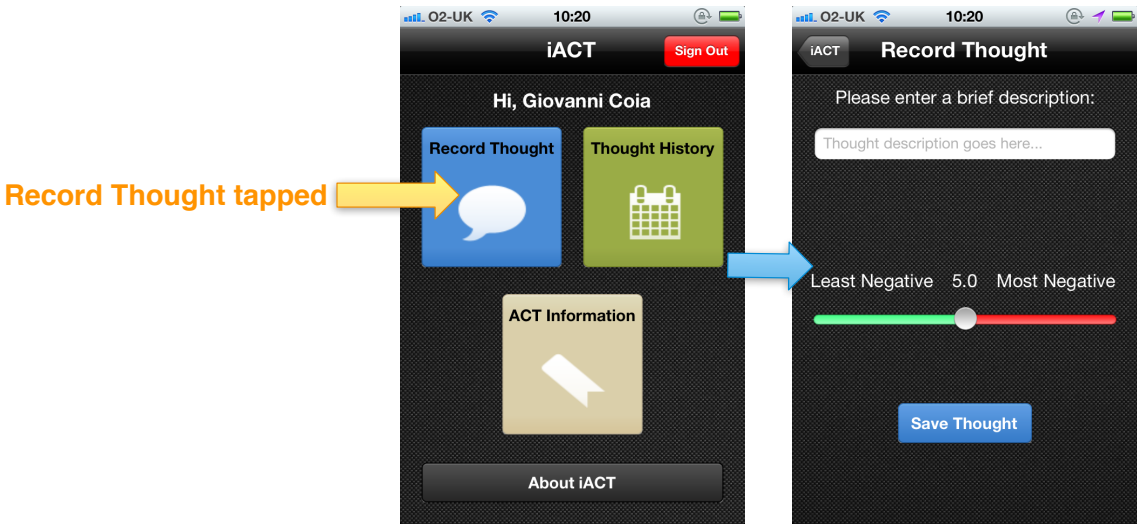


Figure 24 Main Menu and Record Thought View

Selecting the 'record thought' option from the main menu initiates the process of recording a thought as illustrated by *figure 24*. At this point a couple of data structure classes are instantiated. These classes use the Core Data ORM to map the thought data against new records inside the application model. Both a thought object and an occurrence object will be created. The thought object contains the description of the thought as a string, and its child occurrence object contains its attributes such as time recorded, rating and location.

The first step in recording a thought is to start logging the user's location. As it can take a few moments to acquire a GPS location it is good practice to start recording the location as soon as possible. This is to ensure

that when it is time to save the data the iPhone will have acquired an accurate enough fix on the user's location. Location recording takes advantage of iOS's Core Location framework. This provides a full set of software tools for using the iPhone's GPS hardware to record the handset's present location. Location recording works by instantiating an object of the CLLocationManager class. One then sets a distance filter attribute that tells the location manager how often to update the location. As I am only looking for the present location, I set the location manager to only send updates if the user moves more than 200 meters while location recording is taking place. Setting a smaller distance results in a higher frequency of GPS updates which in turn can negatively affect battery life. CLLocationManager will then begin sending location update messages to the record thought controller. This contains some interface methods to handle these updates and record the location.

Following this the next step is to record the user's input. The user enters the content of their thought as a string at the top of the screen. A simple slider is then used to set a rating for this thought, which will ultimately be stored as a numerical value. When the user clicks save a number of actions take place. First, the thought description field is validated to ensure the user has actually typed something. If they have not, a message is displayed prompting them to do so. If a description is present this is then searched for in the model. This is because if it already matches an existing thought, I want to record a new occurrence only. If the thought has never been entered before both a new thought and occurrence are created. The final stage is shut down the CLLocationManager as the application no longer requires location updates. If the input is valid the newly created thought data is passed to the next view in the record thought sequence by means of a segue to the thought manipulation view.

4.5.4 Thought Manipulation View

When this view loads the thought description is taken from the thought object that has just been passed from the record thought view and displayed as a text label on the screen. A random colour is then chosen for the background. This is to help change the context in which the thought is considered. As per the client's design requirements, the user is then able to interact with the visual depiction of the thought.

These interactions are primarily achieved through the use of iOS's gesture recogniser classes. These provide ready-made functionality for recognising common multi-touch gesture types such as tap, pinch and rotate. There are three basic interactions that this view facilitates. The first is animating the movement of the label from one position to another. This is achieved through the use of iOS's Core Animation framework. When the user taps the screen, the view then moves the label to the location that was just tapped. The pinch gesture results in the scale of the label being adjusted in proportion to the size of the pinch. Finally, a rotating gesture will rotate the label by the same number of degrees as the rotation gesture.

Once the user has finished manipulating the on screen display of the thought content it is time to perform the final rating. To do this, the user taps the done button in the upper right corner. This performs a segue to the final view of the thought recording process. Once again, the thought data is passed on with the prepareForSegue method.

4.5.5 Final Rating View

Having interacted with the thought, the user is presented with a similar view to the initial thought recording screen. This simply shows the exact same rating slider again, and once the user has finished they tap a save button. Upon tapping save, several events take place. The thought and occurrence objects are both saved to the Core Data model. A RestKit REST message is then sent to the web site containing the newly recorded thought data. The iPhone sessions controller on the web server then adds this to its own data model. This is performed in a background thread to allow the user to continue using the application while the communication is taking place. A segue to return the user to the main menu then occurs.

4.5.6 Thought History View

The thought history view is one of the most complex controller and view combinations in the application. This takes data from the Core Data model, selecting only records for the logged in user, and formats the results for display in a table. Tables are one of the most common iOS user interface elements and are present in almost every iOS application. iOS provides a framework for their creation by means of a default view controller class known as a UITableViewController. By conforming to the specification for the UITable interface, this class provides default methods for populating the table. One adds code to link to the data model in these methods.

Each row in the table is known as a cell. A table can have multiple types of cell that can be separated with headings indicating the content of that section. For example, the address book app typically separates groups of cells alphabetically, with a heading for each letter. Cells can be populated with many kinds of data. In the thought history view, there are two types of cell as illustrated in *figure 25*. The first is the 'record new thought' cell. Pressing this simply segues to the same record thought interface that can be accessed from the main menu. This always appears at the top of the table. This is simply to give the user quick access to this feature from the thought history screen, in case the thought they were looking for is not present.

Figure 25 UITable Overview

The second cell type used is a cell representing a single thought in the data model. Each cell has a label displaying the content attribute of that thought. Selecting that particular cell will perform a segue to a detailed overview of the thought, complete with access to its occurrence data. Both cells have a right pointing disclosure arrow on their right edge to visually indicate to the user that tapping that cell will provide further information.

As well as providing an overview of the thoughts the user has entered, the thought history view also facilitates downloading new thoughts entered into the server by the user's therapist. Originally I had provided this functionality as a button, but as the project progressed I moved to a 'pull-to-refresh' style interface. Although pull-to-refresh is surprisingly not an official iOS interface element and cannot be found in any Apple software, it has rapidly become one of the most common interface elements in iOS. It is extremely common for third party applications with table views to use this gesture to facilitate refreshing data, and it can be found in some of the iOS's most popular applications such as Twitter and Facebook. Although difficult to explain in words, it provides an interface that feels surprisingly natural to the end user.

Pull-to-refresh interactions work by making the top of the table function as if made from elastic. As the user begins to pull down a "Pull down to refresh…" message appears as shown in *figure 26* alongside an arrow indicating the direction of travel. Should the user pull far enough the text changes to "release to refresh…" and the arrow rotates, indicating the direction of release. Upon releasing the gesture a sync action is triggered. While the sync is in progress the table stays pulled down and a progress wheel spins. Once the sync completes the table's elasticity appears to release, returning the top of the table to its usual position and

indicating to the user that the sync has completed. If the user changes their mind about syncing, simply releasing the pull gesture before reaching the "release to refresh…" message will cause the table to spring back to its default position and no sync takes place.

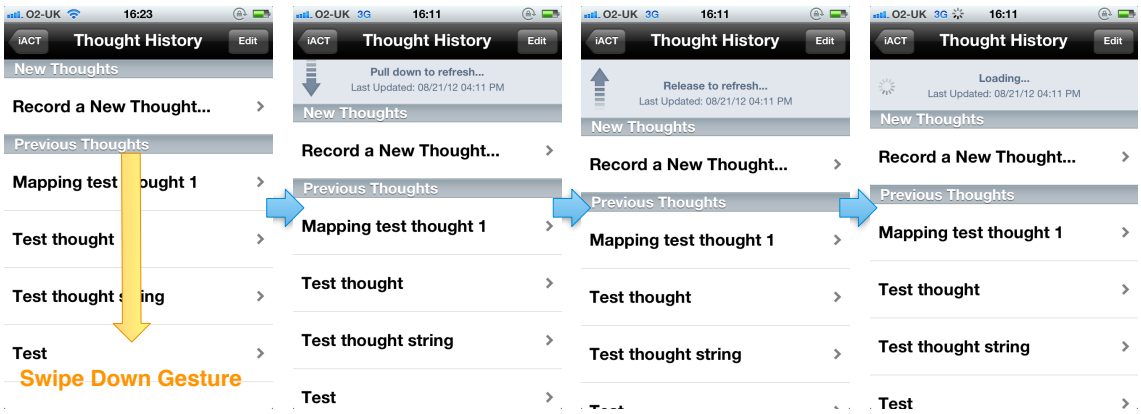


Figure 26 Pull-to-refresh Overview

Although there is no explicit visual cue that this function exists, because it has an elastic feel the user will normally discover it simply by flicking through the table. Further, this style of interaction is now so popular that many users will likely try to do this instinctively. It is widely assumed Apple's reluctance to adopt this powerful user interface paradigm in their own applications is due to Twitter having patented the concept.[58] However, both Twitter and its inventor, Loren Brichter, have publically stated[59] that anyone is free to use this interface element as they see fit, as they will not seek to enforce the patent.

Should the user pull the table far enough to trigger the sync operation, as with all server interactions, a RestKit message is sent to the server. This is sent along with the user's ID number. Should there be any new thoughts to add to the history these are returned as JSON. The thought history controller then parses this JSON to convert it to new native objective-C thought objects. These are then added to the data model and displayed immediately in the table. The user can then select the newly downloaded thought and start recording some occurrences.

The final functionality provided by the thought history screen is the ability to remove previously recorded thoughts. This is in case the user has made a mistake, or simply no longer wants to keep a record of the thought. Deletion is handled in one of two ways as shown in *figure 27*. Firstly the user can simply click the 'edit' button in the upper right corner. This triggers a deletion button to appear on each cell, allowing that thought to be deleted. Once the user has finished deleting cells they can press the 'done' button, returning the cells to their default state. Alternatively, the user can swipe their finger across the cell. This triggers a deletion button to appear in that cell, pressing this resulting in the destruction of that data. A user can simply tap anywhere else on the display to dismiss this option if they change their mind. Both of these interactions are standard deletion gestures for a UITable, and can be found in many other applications. This increases the likelihood of the user being familiar with this style of interaction.

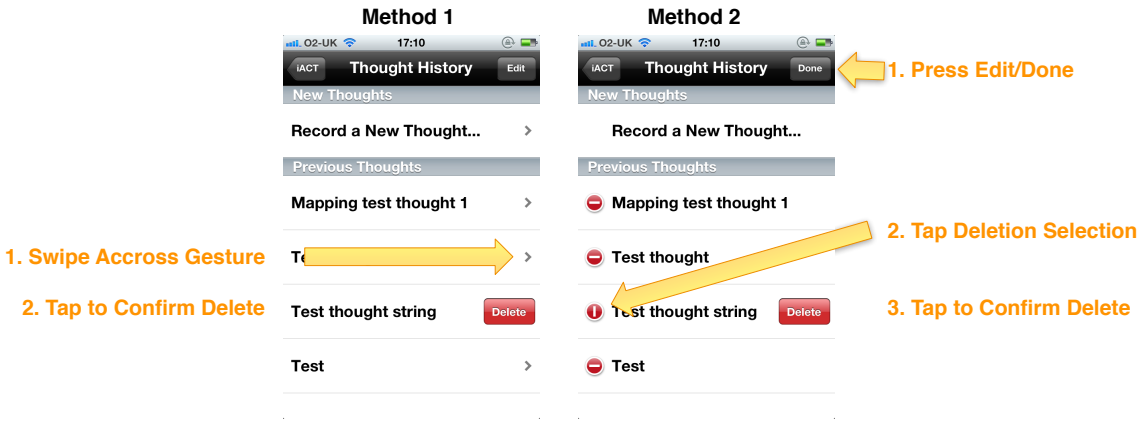


Figure 27 Deletion Interaction Methods

Once the delete button has been pressed the thought is deleted from the Core Data model. A message is also sent using RestKit to the server to delete this thought from the server's database. This allows the user total control over what is recorded. If a user later decides they would rather their therapist not see a particular thought, they have the ability to remove it.

4.5.7 Thought Occurrence View

Selecting a cell on the thought history view performs a segue to the thought occurrence view, as illustrated in *figure 28*. This view provides an overview of the selected thought with a map of all the locations in which it was recorded, along with a UITable displaying a list of occurrences. This list is populated using the occurrence records for the selected thought from the Core Data model.

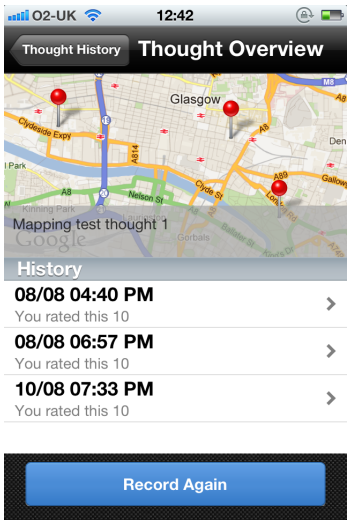


Figure 28 Thought Occurrence View

When this view loads, its first task is to display a map. Mapping in iOS is one of its most powerful features, accessible through a framework called MapKit. Using MapKit, one can interact with Google Maps to display map or satellite imagery, as well as providing information overlays. In this view an overlay containing pins representing the location of each occurrence is displayed. A helper method in the view's controller obtains the maximum latitude and longitude coordinates of the occurrence locations to ensure the map zoom is always set to a level that will fit all of the pins. Tapping a pin reveals the occurrence's date and rating.

Underneath the table of occurrences a 'Record Again' button is always displayed. This allows the user to quickly add a new occurrence of this particular thought. A segue will transition to the record again view. This view is identical to the record thought view with the exception of the thought description field. As this has

already been entered previously there is no need to type it again, so a label showing the description appears in place of a text entry box.

Much as with the thought history view, tapping a cell in the list of occurrences will segue to a new view. This view displays a larger map showing only the selected occurrence, and underneath lists some of the occurrence's attributes. The user can then tap a back button in the UINavigationBar to return to the thought occurrence overview.

4.6 Xcode Issues

Although working with Xcode was a largely positive experience, there were a number of issues I encountered whilst writing the code for the application. These ranged from software bugs to unusual coding styles. The terminology used in the development of iOS software is also somewhat confusing in comparison to other object orientated development environments.

4.6.1 Nomenclature

One of the biggest issues in working with iOS is that many of the common object orientated software development terms are called entirely different things. For example, interfaces are called protocols, method calling is known as message passing and instance variables are called properties. This initially made development a little confusing, but once one begins to understand that these are in fact just different terms for the same things one encounters in any other object orientated language the problem quickly went away. The confusion is further compounded however in that many of the textbooks are written by authors who have experience in multiple programming languages, and they can often start to use the terminology interchangeably by accident. Additionally there is a whole host of concepts that are unique to iOS development, such as the use of Outlets and Actions when connecting the application views to a controller.

Another interesting aside was that primitive types can exist either as primitive types or objects depending on the context in which one is using it. An integer can therefore be stored both as an integer and an 'NSNumber'. For example, there is very little static typing used in iOS method signatures. Methods frequently use the generic object type 'id' and use either casting or introspection to use or determine the object's nature. As a result, methods that take generic objects as a parameter often require the user to 'package' primitive types inside an object, rather than passing the primitive type itself. While it is possible to type check the parameters, this does not seem to be a style of programming that is used widely, either by developers or by Apple's APIs. In some respects this may help in the reusability of the code. For example, one could change a button in the GUI to a switch, and because the method behind the button is using generic types, rather than specifying a UIButton, it is likely the code will work without change. However, my only prior programming experience has been Java, in which type checking just about everything explicitly is the norm, making this an unusual style to work with.

Another issue is that while almost all of iOS's libraries are now object orientated and can be worked with in objective-C, not all of them are. Some of the older frameworks, such as the address book framework, are still written in C. This means that one cannot treat address book data as native objective-C objects. One instead must convert between the two using bridged casts. This can be a little jarring as the syntax suddenly changes halfway through a method. Much of the graphics APIs are still written in C too. It can make for some pretty ugly code at times.

4.6.2 Bugs and Coding Style

One unexpected issue in developing for iOS is that because so much of the functionality of your application is provided ready built through iOS's extensive framework libraries, one does encounter bugs in their implementation. For example, Apple's Core Image API for applying filters to image files does not work as specified in the official documentation. Both Xcode and the iOS are in a constant and rapid state of development, and some of the issues I encountered were actually fixed in software updates over the course

of the development period. Xcode's feature set at times feels almost liquid given the rapid rate of bug fixes and new features it receives. A good example of this was in how one handles arrays. In many languages one can access a specific array index with syntax similar to the following:

```
x = arrayName[2];
```

Typically this will return the item at array index 2 in languages such as Java. This was however initially not possible in objective-C. Instead, one had to use the vastly more verbose:

```
x = [arrayName objectAtIndex:2];
```

This is exacerbated when one discovers that there is no support for arrays of primitive types in iOS. Notice the above syntax uses the message call objectAtIndex. As this suggests, NSArrays may only contain objective-C objects, not primitive types. This means one must first package a primitive type inside an objective-C object before adding it to the array. One must then extract it again every time that array index is called if you want it returned as a primitive type. For example, a primitive number such as a double or an integer would have to be packaged inside an NSNumber object to be stored in the array. In theory one can work around this by relying on older C style arrays, which are still supported, but this makes for some very messy code. This means adding the integer 1 to the array requires two method calls and looks like this:

```
[arrayName addObject:[NSNumber numberWithInt:1]];
```

And getting it out again also two method calls:

```
int x = [arrayName objectAtIndex:2] intValue];
```

This may not seem like a huge difference, but it quickly becomes very irritating having to type out one or more nested method calls every time you want to perform a simple array index access. Whilst developing the project Xcode 4.4 was released, adding support for the much shorter Java style syntax. Ultimately this was too late for me to consider going back and refactoring my code to take advantage of the simpler syntax, but it is a nice improvement. This was possible not because of any improvements to the underlying language, instead Apple have made the compiler smart enough to convert the shorter syntax into the longer syntax with method calling behind the scenes. Apple often choose to try to improve short-comings in the objective-C language in this way by making the IDE or compiler smarter, rather than fundamentally changing the underlying language. It is this approach that similarly enables the important new Automatic Reference Counting feature, of which this project takes advantage of, to work.

4.7 Testing Overview

Given this project required the development of a web application and an iPhone application across a tiered system architecture, testing of the system required several different approaches. In this section I will explain the testing strategy used in each component of the system.

4.7.1 Testing the Web Site

As outlined in the implementation overview, testing of the web site was made very easy by adopting a test-driven development process. As I was writing tests before implementing functionality I always had an up to date suite of tests to ensure the site was working as required. This was facilitated with the RSpec library.

Writing tests in RSpec is surprisingly simple. Written entirely in Ruby, one simply sets out some conditions that must satisfied for the test to pass. The test suite is then executed at the terminal prompt; should any test fail an appropriate error with details of the failing test is shown. A typical RSpec test is shown below in *figure 29*.

```
require 'spec_helper'
```

```ruby
describe User do

  before { @user = User.new(name: "Example User", email: "user@example.com") }

  subject { @user }

  it { should respond_to(:name) }
  it { should respond_to(:email) }
end
```

Figure 29 RSpec Test Example

This simple example test is to validate the user model. The 'before' line creates a new user object in the model and the remaining lines check that it has been created with the appropriate attributes. Using the test first approach, one would write and run the above, noting the failure. After adding the appropriate functions to the data model the test is re-run, hopefully passing this time. As the site was under git source control it was very easy to roll back mistakes or create test branches for new features.

As well as using the RSpec library I also used the terminal to test interactions for the website. While the server is running Rails will list all incoming HTTP requests in the terminal window. Rails then lists the controller used to generate the response along with any SQL used to access the database. If an error occurs or an incorrect SQL call takes place an easy to understand error message is displayed. For each possible interaction with the site I confirmed that the appropriate response took place in the terminal window.

Rails also usefully permits the launching of the site in a developer mode. In this mode one can configure each page to display useful data about how it was constructed. Using some simple CSS styling, one can get this data to appear neatly at the bottom of each page as demonstrated in *figure 30*. This allows for a quick check of the controller, method and parameters used to construct the presently shown view.

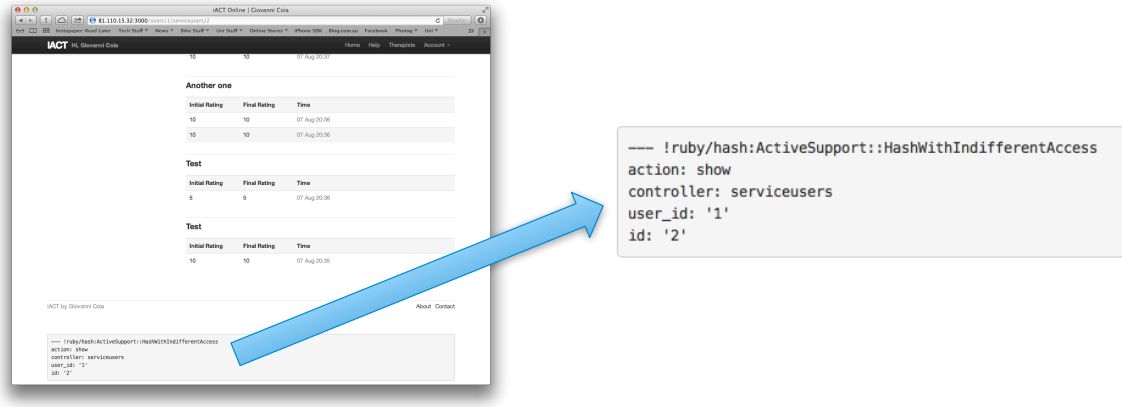


Figure 30 Debugging Information

The next stage of the testing process was to perform some browser testing. As I had tried to remain fully standards compliant throughout the design of the web site I was confident that there would be few, if any, issues with rendering the site correctly. To confirm this I accessed the site using the latest versions of many popular browsers. To this end I checked the site in *Chrome*, *Firefox*, *Internet Explorer*, *Opera*, *Safari* and *mobile Safari*. As expected, the page rendered correctly in each.

4.7.2 Testing the REST API

As the API is fundamental to how the application works, I needed a quick way to test it without having to construct a client application or web site. Given that a REST request is simply an HTTP request with relevant parameters, this could be done from the terminal. However, to make this process easier I discovered a Firefox plugin called *Poster*[60] that permits the construction of HTTP requests and then provides a text output of the

response. This allowed me to quickly validate all of the API's methods before constructing the iOS application. Each of the REST request types supported by the API was constructed in Poster. For each request type a succeeding and failing case was constructed. This confirmed that the API worked as expected.

4.7.3 Testing the Data Models

As both the mobile application and the web site used an SQLite 3 database to store the model this meant ensuring that data was being validly recorded could be achieved with the same method. After adding a new record to the database I would open the SQLite files using a program called SQLite Data Browser[61]. This application allowed me to easily open the database and visually navigate the database structure in real time as records were being added. This allowed me to ensure that data was being stored correctly.

4.7.4 Testing the iOS Code

The testing process I adopted for the iOS application was not as formal as the one used for the website. Although Xcode does support unit testing, given the huge number of iOS frameworks I had to learn there simply wasn't time to gain an understanding of Xcode's unit testing system. None of the numerous textbooks I used to learn iOS development cover this feature. This meant I would have had to learn how to use it solely from blogs and developer community forums. Given that the data model and controllers ended up changing quite fundamentally in how they operated it is not clear that unit tests would have been of much use during the implementation process. Now that the data model is Core Data based it would however be much easier to design a reusable set of unit tests in the future.

Despite this, I was able to rigorously test my application using Xcode's debugging tools and a black box testing methodology. Following the black box approach, with each new version of the software I would test every feature of the application and check that it behaved as intended. If anomalies or errors were discovered I would then use the debugging tools. A full listing of the black box tests used can be found in the documentation following this chapter. Xcode usefully allows the programmer to insert break points into the code while the application is running. This combined with Xcode's ability to view the contents of variables in real time by highlighting them in the code meant I was able to comprehensively debug any issues that arose during development.

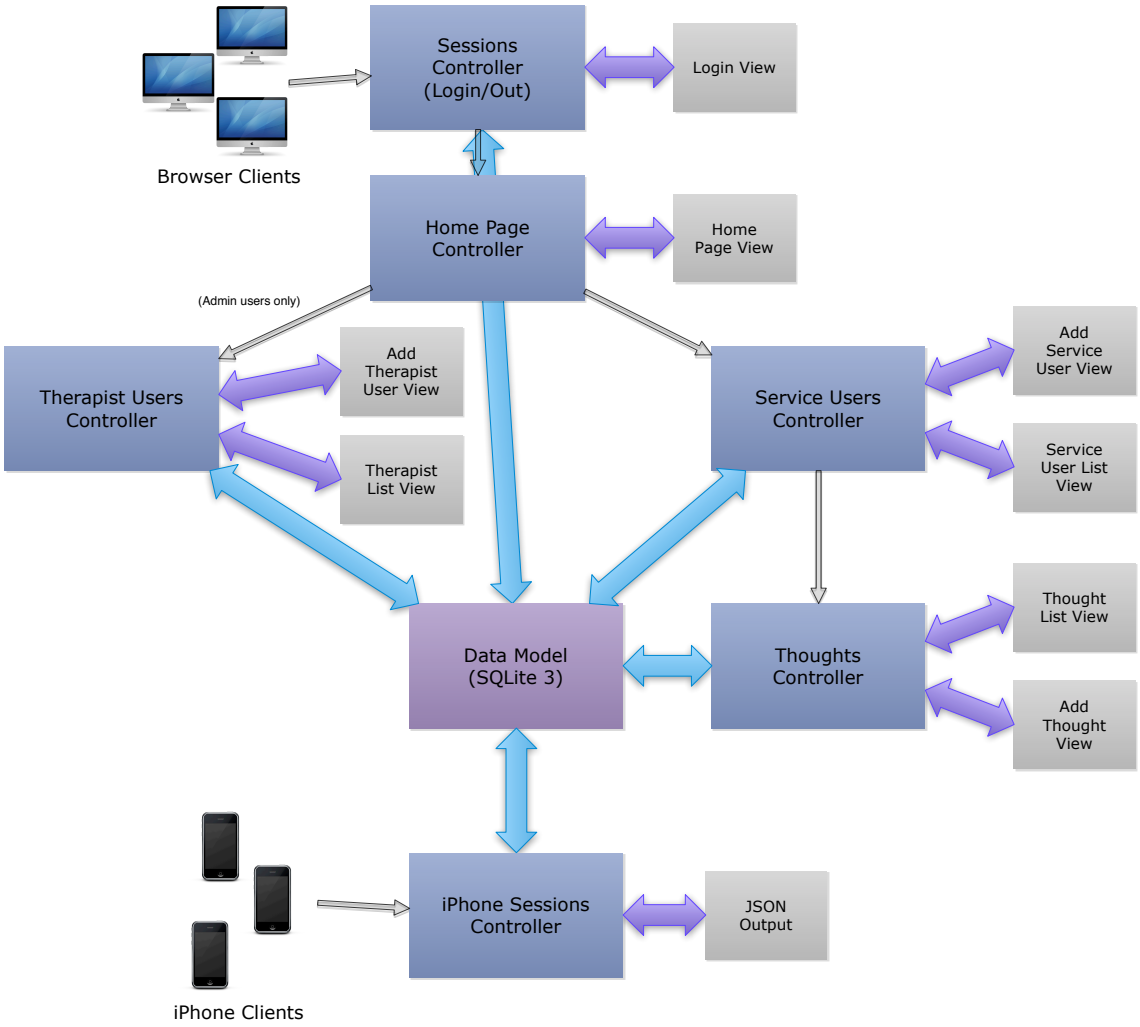
Another testing feature I used was Xcode's 'zombie objects'. Because of the way in which memory is managed in iOS, the standard terminal output on a crash due to a null pointer exception is unable to express precisely which null object was called. To help debug such scenarios Xcode provides a useful feature known as zombie objects. This effectively stops any object from being removed from memory. Xcode is then able to display a detailed error explaining that a message was passed to an object that would otherwise have no longer been in memory. One however must be careful when using this feature, as it can quickly cause memory leak issues.

4.8 Documentation

This section outlines the documentation for the iACT system. It should be noted that producing documentation in Xcode is one of its poorest features. There is nothing comparable to the excellent JavaDOC API one typically finds in Java IDEs. For the creation of the iOS application documentation I used a popular third party utility called doxygen[62]. This freely available tool can generate documentation in a similar format to JavaDOC from Xcode source files. One simply comments the code in a special format. Doxygen then recognises these comments and automatic documentation is generated.

Documenting the web application was much easier as once again I was able to exploit the benefits of using a WAF. I installed a documentation-generating gem called YARD[63]. Documentation was then created by a single word command at the terminal. Full documentation for both the web application and the iOS application can be found in electronic format accompanying the source code for each.

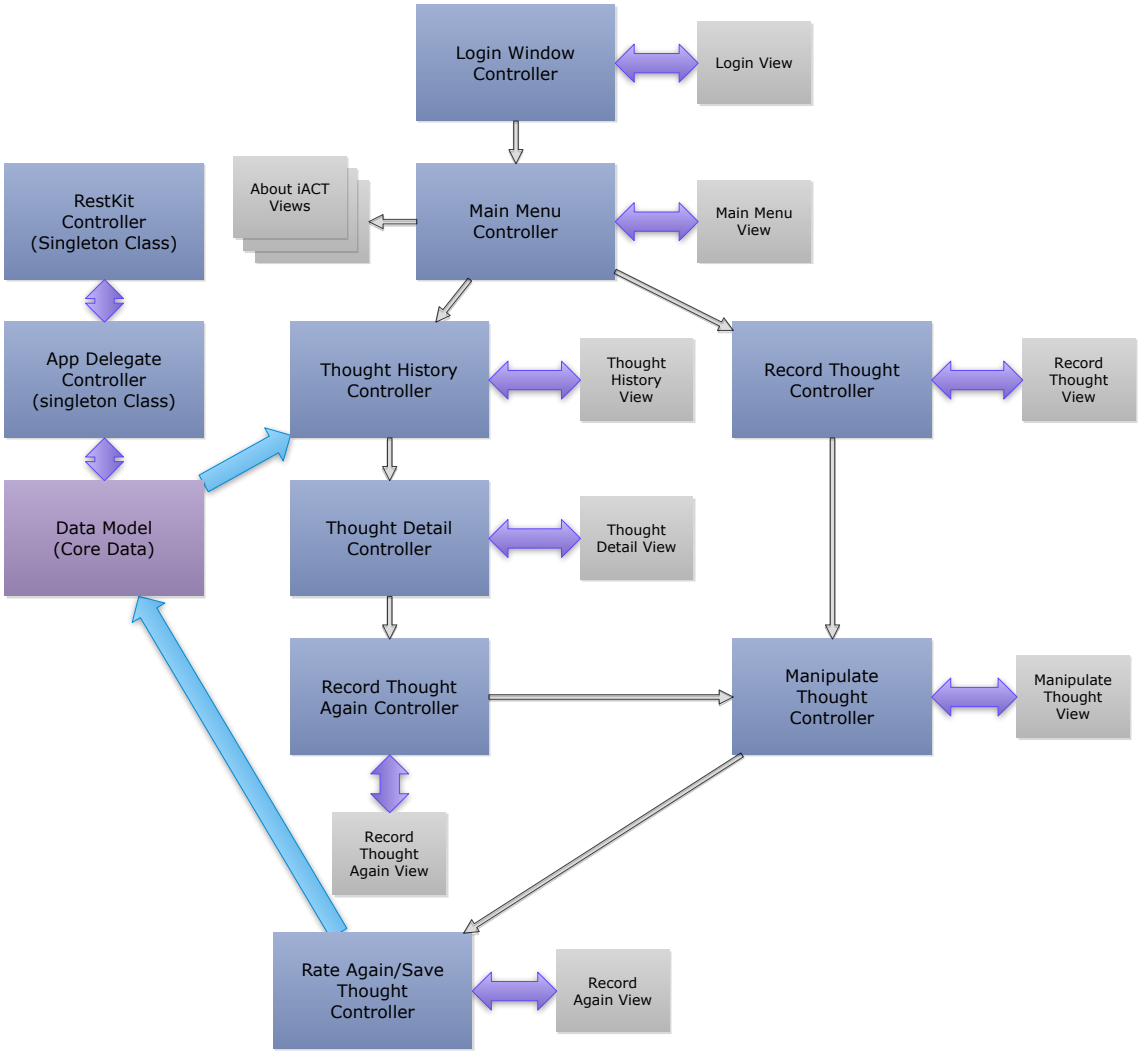
4.8.1 iACT Online Website Design Overview



iAct Web Application Class Documentation

Please see electronic documentation accompanying source code.

4.8.2 iACT iOS Application Design Overview



iACT Class Documentation

Please see electronic documentation accompanying source code.

4.8.3 Black Box Testing Results

Functionality Test	Expected Result	Actual Result
Sign in view rejects invalid login credentials when 'Sign In' button tapped.	If user enters invalid login credentials an error message should be displayed.	As expected
Sign in view accepts valid login credentials when 'Sign In' button tapped.	A segue to the main menu view should be performed.	As expected
When 'Sign Out' button tapped user should be logged out of the application.	A segue to sign in view. No access to user data.	As expected

Navigation bar interface should be present once logged in.	A standard iOS navigation bar should permit navigation through the application's views.	As expected
Main menu view should obtain user's name from server and display it.	'Hi, [user name]' should appear at top of menu.	As expected
If a user is logged in, a relaunch of the software after quitting or force quitting should not log user out.	User remains logged in even after quit or force quit of application.	As expected
Main menu view should present correct list of functions.	'Record Thought', 'Thought History', 'ACT Information' and 'About iACT' buttons should be displayed.	As expected
Record thought function should segue through three views, first rating, manipulation and final rating.	The appropriate three views should appear sequentially.	As expected
Entered thought data should pass from one view to another during thought recording.	Entered thought data should pass to each view.	As expected
Thoughts saved using the record thought feature should be added to the thought history.	Saved thought should appear in the thought history list	As expected
Tapping 'Thought History' button.	Segue to thought history view.	As expected
Tapping 'Act Information' button.	Segue to act information view.	As expected
Tapping 'About iACT' button.	Segue to about iACT view.	As expected
Tapping the 'Send Feedback' button on the about iACT view.	A prepopulated email window should appear, with 'iACT Feedback' as the subject and my email address in the To field.	As expected
Tapping 'Record a New Thought…" cell on the thought history view.	Segue to the record thought view.	As expected
Tapping a previously recorded thought in the thought history view.	Segue to thought detail view, showing location of occurrences and a list of occurrences.	As expected
Tapping a map pin in the thought detail view.	Date and rating of occurrence displayed.	As expected

Tapping an occurrence cell in the thought detail view.	Segue to map showing location, with information overlay.	As expected
Swipe gesture across thought cell in thought history view.	Option to delete thought presented.	As expected
Tapping 'edit' button on thought history view.	Option to delete thoughts presented. Button title changes to 'done'.	As expected
Pressing 'delete' on a thought cell.	Deletes the selected thought, removing it from thought history.	As expected
Pull-to-refresh gesture on thought history view.	Adds any newly added thoughts from the web server to the thought history.	As expected
Tapping a point on the ACT Information hexagon.	Segue to view containing information about that ACT principle.	As expected

[43] Hartl, M., 2012. *Ruby on Rails Tutorial: Learn Web Development with Rails*. [Online] Available at: http://ruby.railstutorial.org [Accessed 01 June 2012].

[44] RSpec, 2012. *RSpec*. [Online] Available at: http://rspec.info [Accessed 05 June 2012].

[45] legislation.gov.uk, 1998. *Data Protection Act*. [Online] Available at: http://www.legislation.gov.uk/ukpga/1998/29/contents [Accessed 14 June 2012].

[46] Twitter, 2012. *Bootstrap*. [Online] Available at: http://twitter.github.com/bootstrap/ [Accessed 10 June 2012].

[47] mislav, 2012. *will_paginate*. [Online] Available at: https://github.com/mislav/will_paginate/wiki/ [Accessed 19 June 2012].

[48] bcrypt-ruby, 2012. *bcrypt-ruby*. [Online] Available at: http://bcrypt-ruby.rubyforge.org [Accessed 08 June 2012].

[49] OAuth, 2012. *OAuth*. [Online] Available at: http://oauth.net [Accessed 11 June 2012].

[50] Fitzpatrick, J., 2010. *Firesheep Sniffs Out Facebook and Other User Credentials on Wi-Fi Hotspots*. [Online] Available at: http://lifehacker.com/5672313/sniff-out-user-credentials-at-wi+fi-hotspots-with-firesheep [Accessed 18 June 2012].

[51] Butler, E., 2010. *Firesheep*. [Online] Available at: http://codebutler.com/firesheep [Accessed 19 June 2012].

[52] Ray, J., 2012. *Sam's Teach Yourself iOS 5 Application Development in 24 Hours*. 3rd ed. Sams.

[53] Mark, D., Nutting, J. & LaMarche, J., 2011. *Beginning iOS 5 Development: Exploring the iOS SDK*. 1st ed. London: Apress.

[54] Stanford University, 2011. *iPad and iPhone Application Development*. [Online] Available at: http://itunes.apple.com/gb/course/developing-apps-for-ios-fall/id495054839 [Accessed 01 February 2012].

[55] Callahan, K., 2012. *Accessorizer*. [Online] Available at: http://www.kevincallahan.org/software/accessorizer.html [Accessed 29 June 2012].

[56] Wikipedia, 2012. *Serialization*. [Online] Available at: http://en.wikipedia.org/wiki/Serialization [Accessed 09 June 2012].

[57] RestKit, 2012. *RestKit*. [Online] Available at: http://restkit.org [Accessed 29 May 2012].

[58] Brichter, L., 2010. *US Patent & Trademark Office*. [Online] Available at: http://appft.uspto.gov/netacgi/nph-Parser?Sect1=PTO2&Sect2=HITOFF&u=%2Fnetahtml%2FPTO%2Fsearch-adv.html&r=1&f=G&l=50&d=PG01&p=1&S1=20100199180.PGNR.&OS=dn/20100199180&RS=DN/20100199180 [Accessed 19 August 2012].

[59] Brownlee, J., 2010. *Twitter Vows Never To Weaponize Their Patents, Including Pull-To-Refresh*. [Online] Available at: http://www.cultofmac.com/161367/twitter-vows-never-to-weaponize-their-patents-including-pull-to-refresh/ [Accessed 19 August 2012].

[60] Milowski, A., 2012. *poster-extension*. [Online] Available at: http://code.google.com/p/poster-extension/ [Accessed 20 June 2012].

[61] SQLite Database Browser, 2012. *SQLite Database Browser*. [Online] Available at: http://sqlitebrowser.sourceforge.net [Accessed 10 June 2012].

[62] Doxygen, 2012. *Doxygen*. [Online] Available at: http://www.stack.nl/~dimitri/doxygen/ [Accessed 05 August 2012].

[63] YARD, 2012. *Yay! A Ruby Documentation Tool*. [Online] Available at: http://yardoc.org [Accessed 11 August 2012].

5. Evaluation

During the proposal period it was agreed that iACT would not be trialed with 'real' users during the project development phase. This was largely due to the complex legal, ethical and security issues surrounding capturing highly sensitive personal information from individuals in a real therapy environment. The aim of the project was therefore to build and implement a system that could one day form the basis of a trial. Given that the system must ultimately handle many users accessing the service simultaneously, it was still necessary to perform a small degree of user testing.

To this end the system was deployed on the iPhones of several friends and family, as well as the handset of the client. Some testers were also given accounts to use the web application as well. It was made clear to each tester that I did not intend for them to use the application to record actual thought data. To ensure I did not handle sensitive personal information I asked that any data submitted through the application simply be test sentences. The feedback from these users was very positive, with no real issues uncovered save for one small fault with the deletion of a thought syncing back to the server. This was the result of accidentally leaving a test value hard-coded in the iPhone application. Once this was amended to use the correct value thought deletion worked correctly for all users.

Given the decision not to trial the software with real users at this stage the primary basis of this evaluation has been on testing the finished product against the client's requirements specification and the user feedback. From this testing and my own analysis, the following evaluation of the software was prepared. Evaluation of iACT's effectiveness as a tool for ACT was not performed at this stage. The client does however hope to run a clinical trial in the future to ascertain this.

5.1 Evaluation Against Project Objectives: iOS Application

The finished project has delivered a fully functional and near feature complete iOS application. Here I provide an evaluation of its finished features against the objectives for the software. The interface and design for the iOS application, along with the structure of its data model, have remained remarkably true to the original proposal. Most of the required features are also completed and present. For the iOS application the following objectives were originally set:

- Thought recording
- ACT information source
- Randomly timed ACT reinforcement messages

- Exploration of thought with interactive graphics
- Thought history

This section examines the implementation of these key objectives as well as several others that became important as the project progressed.

5.1.1 Thought Recording

This feature is fully implemented as per the client's requirements. A user is able to enter the content of a thought, rate it and then save it with the recorded thought being sent back to the server. This feature exceeds the minimum specification in that location recording support was also added. When recording a new thought this is checked against previous thoughts to see if it has been recorded before. If so the system automatically generates a new occurrence, rather than create a duplicate thought entry.

As per the client's goals for the software, each thought is rated twice. The first rating is the service user's score for the emotional impact of the thought. After the user has interacted with the thought it is rated again. Both of these ratings are then returned to the web server. This before and after rating will allow the software to be used for research into whether or not measurable improvements in the client's mood can be achieved.

5.1.2 ACT Information Source

Much as in the proposal and the design, an interactive ACT 'hexaflex' appears in the application. Tapping the points of the hexagon provides further information about that ACT principle. Again, this feature is fully implemented.

5.1.3 Randomly Timed ACT Reinforcement Messages

Unfortunately the limited time available coupled with the complexity of implementing push notifications in iOS meant this feature has had to be left out. However, I do have a good understanding of how the technology works and with more time this would almost certainly have been included in the finished software. Apple's API for the push notification service requires an enormous amount of work on the part of the application developer. Messages cannot be sent easily to all users of the software. Instead individual iOS devices must be registered against individual users. Messages must then be pushed one at a time to every user of the service. To do this reliably and efficiently would require a great deal of time, expertise and planning. Many of the resources I consulted regard this as one of the most difficult and painful features to implement in an iOS application. This is so much so that there is now a market for third party services that manage the push notifications on the developer's behalf, such as *Parse*.[64]

Parse works by providing some ready-made code for insertion to your iOS application. This code automatically sends the necessary handset registration data back to their servers. Parse then provide a simple web interface for the sending of push notifications. Whilst I could certainly have adopted a service such as Parse, none of the services I checked supported the random messages iACT required and they were all very expensive to use. These services are clearly designed to facilitate rapid deployment of commercial mobile applications rather than a freely distributed piece of software like iACT. Any attempt to implement this feature will therefore likely require a self-made custom solution.

5.1.4 Exploration of Thought with Interactive Graphics

This feature is present in the finished software. The interactions consist of a randomly chosen background colour and a range of gestures such as taps, pinch and rotation. This allows the user to reposition a string representing the content of the thought on screen. The angle and size of the text can also be adjusted as shown in *figure 31*.

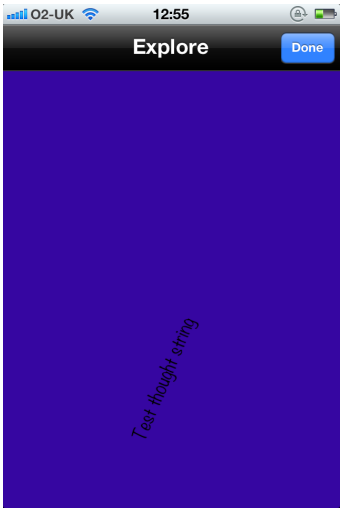


Figure 31 Thought Exploration Interface

This feature calls upon some of the iOS's advanced animation and gesture recognition features, and is one of the key components in providing ACT based interactions in the software. With more development time this is the aspect of the software that I think would be most important to develop further. If the application is to demonstrate a measurable reduction in the emotional impact of difficult thoughts it is this element of the software that must deliver it.

Accordingly it is quite important that this element of the software surprise or delight the user in ways they don't expect. If the user performs the same interaction time after time it seems likely the effectiveness of the interactive graphics will reduce. I was therefore keen to introduce random elements to the code to provide a different experience each time a thought is saved. At present this random element only extends to the randomly selected background colour.

5.1.5 Thought History

This feature is fully implemented and delivers all of the client's required functionality. All thoughts recorded by the user appear in the history listing. I developed the thought history feature a little beyond what had been originally specified in the proposal. As recorded thoughts are represented by two entities in the data model, thoughts and occurrences, it made sense to add a two level navigation interface. Firstly a list of thought all recorded thoughts is displayed. Selecting a thought then displays the second level list of all that thought's occurrences with an option to add a new occurrence. Selecting an occurrence then displays the rating and location of that particular instance.

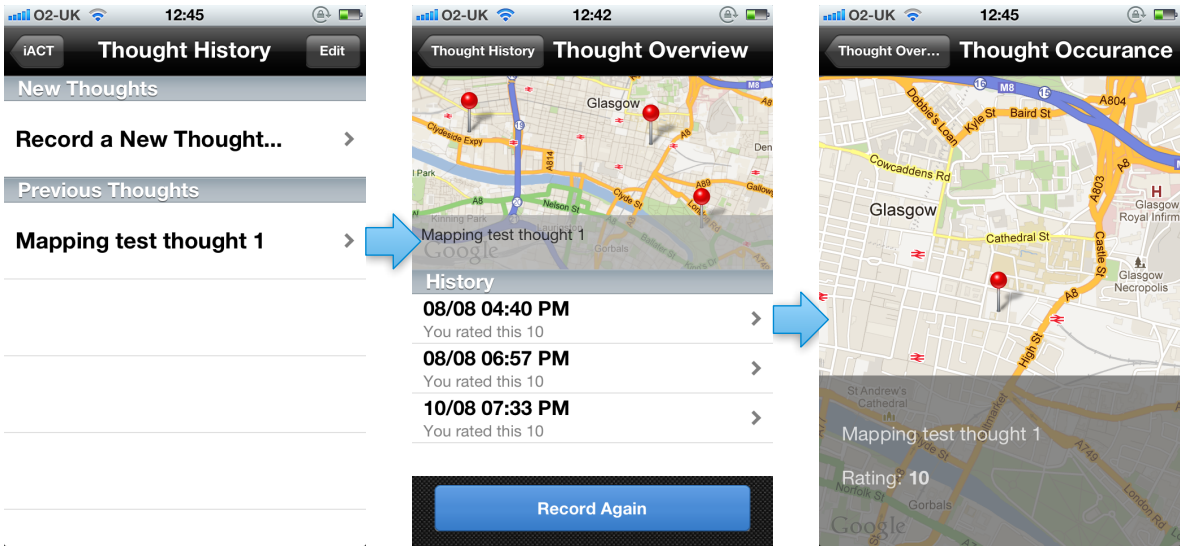


Figure 32 Thought History View

One of the additional features that has been particularly successful is the inclusion of extensive map and GPS features. During discussions with the client we had discussed a concept whereby a user could see if a thought had a particular geographic pattern to its occurrences, helping the user to better understand the root causes of the thought. To facilitate this, when the user selects a thought a map is displayed above the occurrence list as illustrated in *figure 32*. This map has a pin dropped on each occurrence. Tapping that pin then reveals that thought's date and rating. The map automatically adjusts its scale to fit the required number of pins. This easy to use interface adds a new dimension to the viewing of thoughts, with the ability to discover if certain thoughts are clustered around specific areas.

5.1.6 Data Model

This is arguably one of the application's best-implemented features. In the original proposal documentation I only specified a proposed schema for the data, I did not explicitly determine the type of data structure to be used. This was due to my having had no experience in the creation of an iOS application at the time of writing the proposal. The finished application utilises the powerful Core Data ORM framework. This is underpinned by an SQLite 3 database allowing well-structured storage of records with fast search and retrieval of data. The finished database schema is extremely similar to that found in the proposal.

5.1.7 Synchronisation

This is another area in which the iOS application has been very successful. The finished application facilitates two-way communication and sends all of the data types requested by the client. The sending and receiving of data is fast and efficient and the user is kept appraised of all data transmission. The use of the RestKit framework provided sync operations that run as background threads ensuring that the user never sits waiting for a slow sync operation to complete. The user is always provided with a visual prompt when any data transmission occurs, with a spinning wheel appearing in the menu bar. Similarly, when the user updates their thought history feedback is provided to tell them the last time they synced. Working solely with JSON and HTTP status codes over the HTTP protocol ensures that the volume of data sent is as low as possible, meaning that handset battery life or mobile service bill is not unduly impacted.

5.2 Evaluation Against Project Objectives: Web Application

The website, like the iOS application, is fully functional and meets nearly of the goals set in the proposal and the design. For the web site the following objectives were set:

- Web interface for therapist access
- Web server facilitating logging of thought data.

Unlike the iOS application, the finished web site differs slightly from the original proposed design. This was largely because of the decision during development to adopt Bootstrap for user interface elements, meaning I did not create much of the styling myself. I also took the decision to unify the administrator and therapist users, using a boolean to determine which is which. This makes for a more streamlined experience, and made sense given that the adiminstrative users were likely to be therapists themselves.

5.2.1 Therapist/Admin Web Interface Design

The design of the web site has evolved substantially from the proposal. At the time of the proposal I was not really aware of what a WAF was, let alone having any substantial ideas about how to implement a web site. With the iOS application I had some experience in programming that meant I was not completely in the dark about how to proceed. The same was not true of the web site, and it likely for this reason that it has seen such change. *Figure 33* illustrates some of these changes.

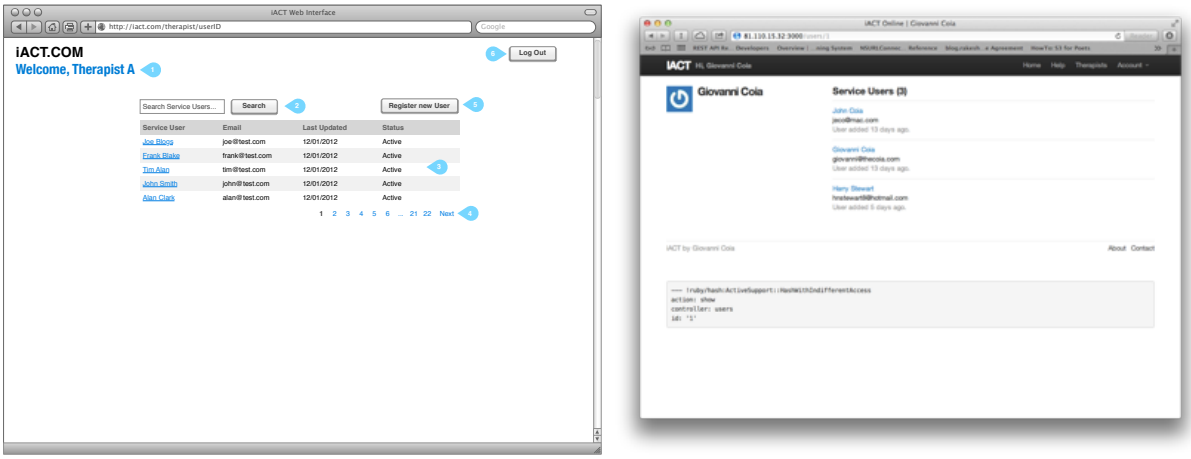


Figure 33 Web Design Developments

The finished site's use of Bootstrap for user interface elements has resulted in what I feel is a very clean and professional look. All of the site's functionality can be accessed at any time through a simple black menu bar along the top of the page. This bar is always displayed and its options change dynamically according to whether the user is logged out, logged in or an administrator. The site also always displays the name of the currently logged in user in the top left, in case multiple therapists use the same computer. This means at a glance a user can make sure they are logged into the correct account.

The site exceeds the original requirements in many ways too. For example, during the original proposal period and discussions with the client it was decided that a security model for the website would not be developed at this stage. Whilst the security model I have created is simple, it is a good starting point for future developments.

The site also provides a great deal more feedback to the user than was originally envisioned. For example, when registering new users, input validation in performed and feedback displayed explaining precisely what is incorrectly entered if necessary. Adding new thoughts also causes a successfully added message to appear on the screen. None of these features are essential, but they greatly improve the site's ease of use.

Although the original specifications for the website did not call for search functionality, it was something I added to my wireframes. Similarly, I had included a date picker to facilitate choosing thoughts by date range. These have not appeared in the finished site due to limits on my time.

5.2.2 Data Model

As with the iOS application, using Ruby on Rails to power the site gave me a powerful ORM system. I did not have worry about writing SQL and could write my view controllers entirely in Ruby. Rails abstracts the code implementing the model from the data storage structure. This means that if the site grows and a new data storage system or software is required, this can be swapped out with one line changed in the configuration file. This instructs Rails to download a new module to interface with the new database software.

The database schema in the finished website is also largely the same as that found in the proposal, except for the absence of an administrative user table. It made more sense as I was developing to incorporate this as an attribute to the therapist's table. As per the client's requirements to obtain anonymous data for future research purposes, the service user's personal data is stored in separate tables from their thoughts and thought occurrences. This means that this table data can be extracted for research purposes easily.

5.2.3 Mobile Application API

The finished API for interaction with mobile clients has evolved enormously from the original proposal. At the time of writing the proposal I had little or no idea how I would set about creating it. The finished API is fast, effective and

simple to use. It could benefit with a small amount of further work to make it fully REST compliant, but this would simply be a refinement, it would not bring further functionality.

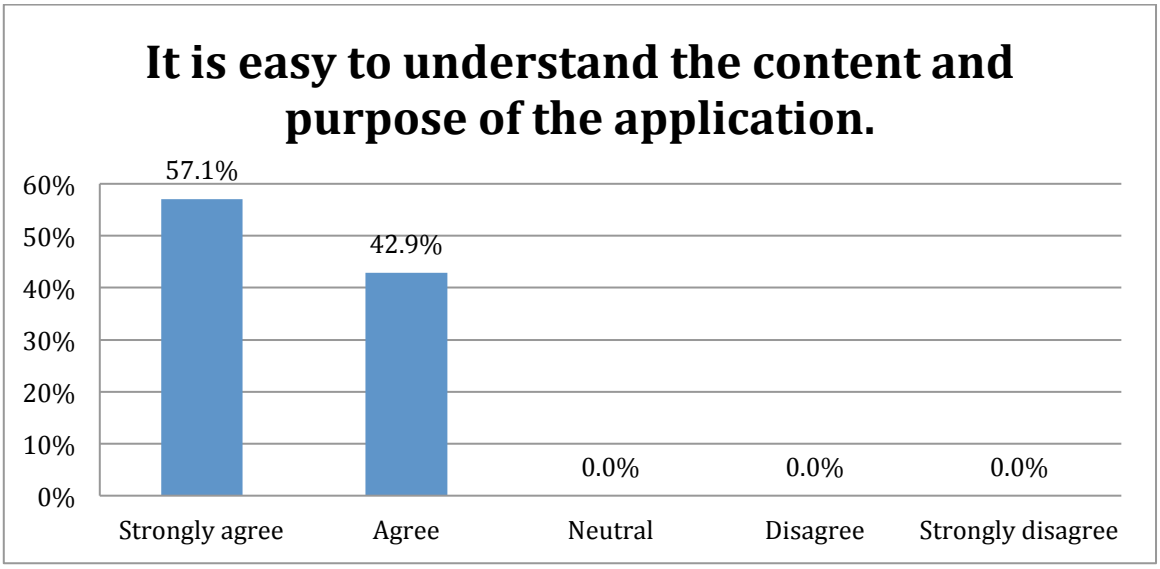
5.3 User Testing

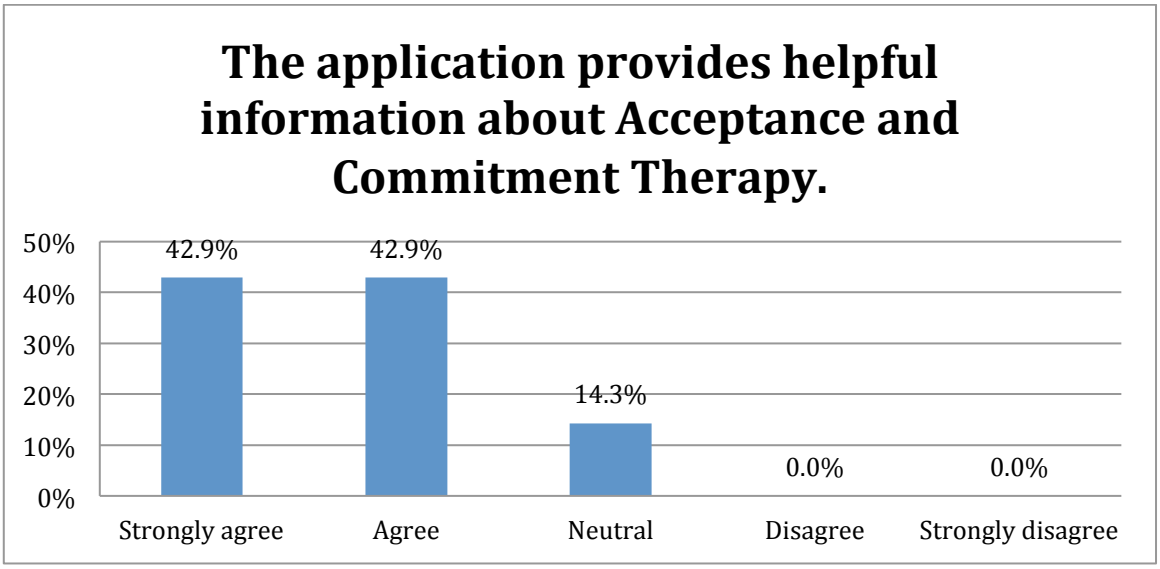
In addition to my personal evaluation of the system against the client's requirements, a small degree of user evaluation was performed as well. This consisted of myself giving a short demonstration of the web site and software coupled with a brief explanation of its purpose. Each user was also given a link to an online survey that asked if they were able to use the key features of the software. In addition the survey also sought opinions on the features users liked best, as well as any suggestions they might have for improvements.

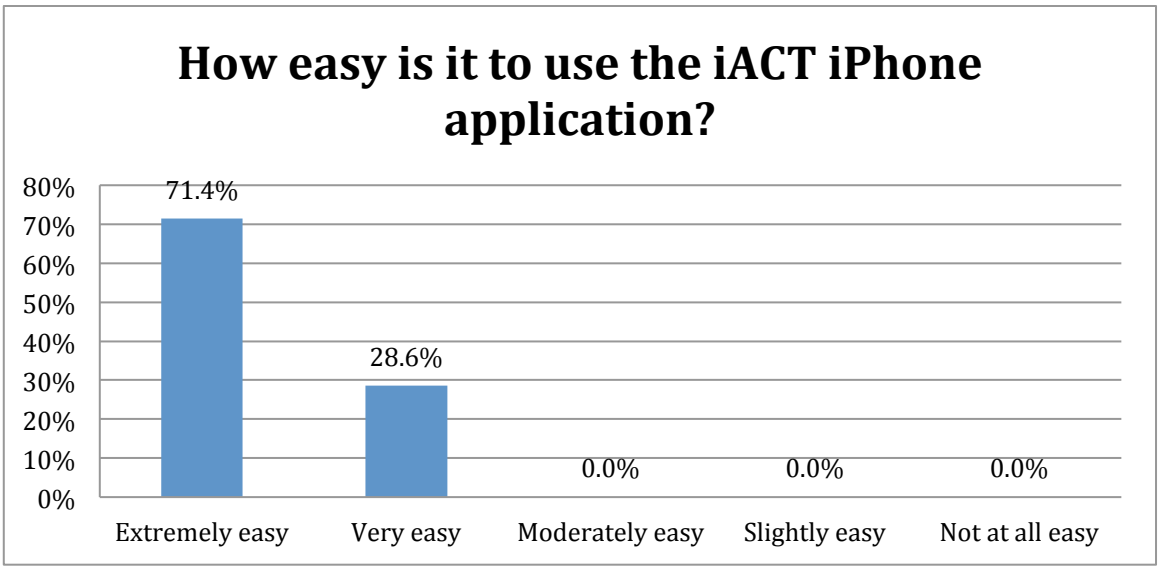
An overview of responses to some of the main questions asked is shown in *section 5.3.1*. Across the board users were generally aware of the existence of the application's key features. Response to the design of the application was also largely favourable, with most users liking the design. Users also found the application provided some helpful information on ACT. Although the sample size is small, and the individuals asked not representative of real ACT service users, some valuable feedback about the software's interface and its ease of use was obtained.

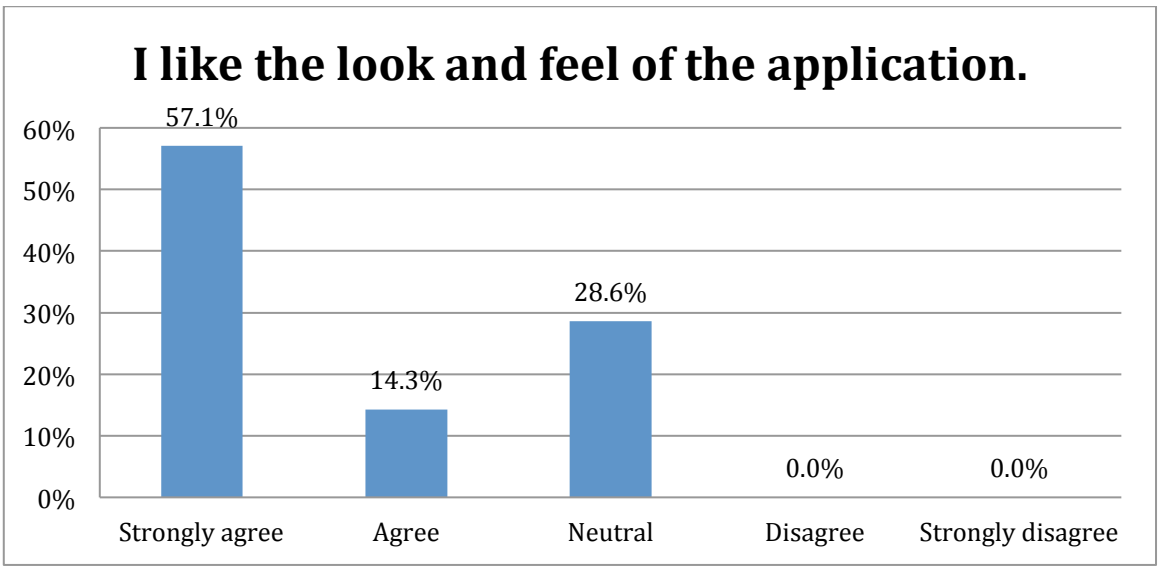
With regard to suggestions left by users, a number of useful points were made. One user noted that the use of a red/green colour for the thought rating slider might prove problematic for colour-blind users. Another user noted that the thought history would be easier to use if most recent thoughts are displayed at the top of the thought history list, rather than the bottom. This kind of ordering is common to most other iOS applications that use table views so I took the decision to change the history order. Having most recent thoughts appear at the top also ensures that the user will always see any newly synchronised thoughts even if their thought history list is extensive.

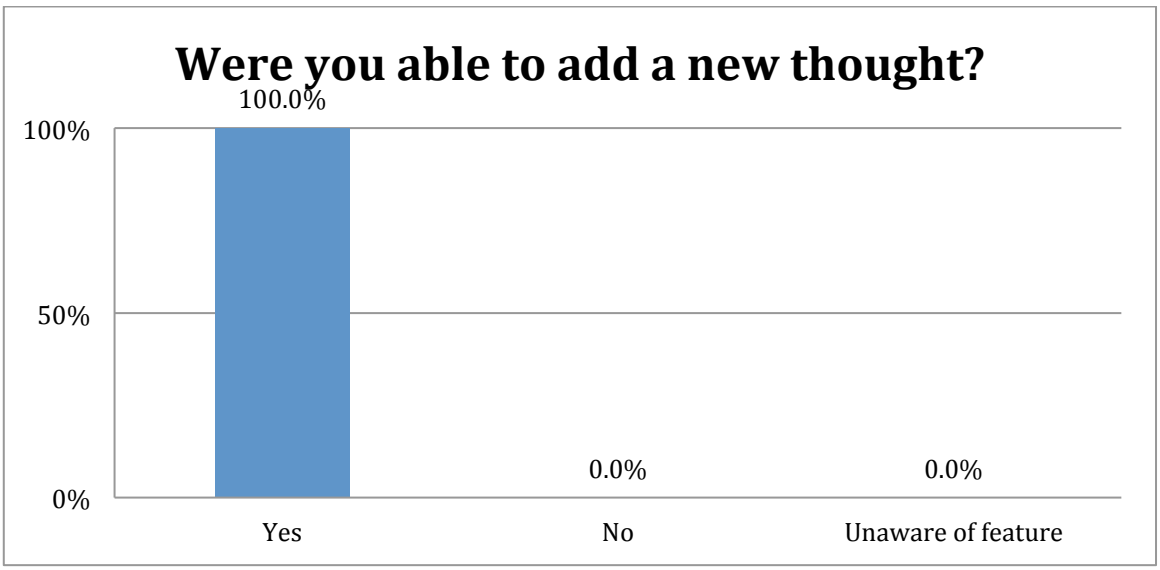
5.3.1 User Testing Data

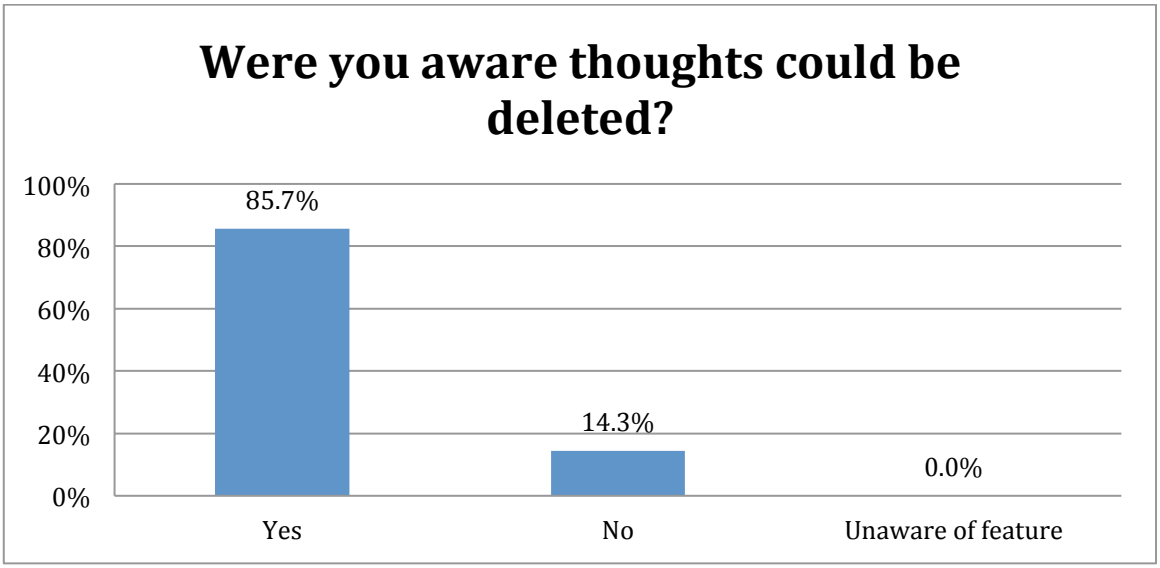


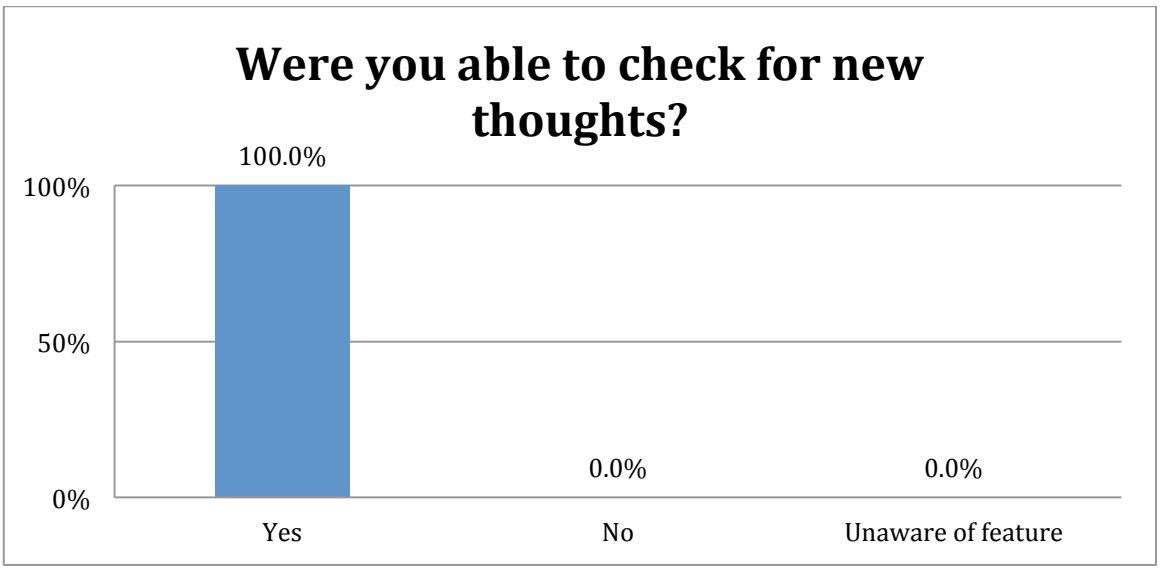


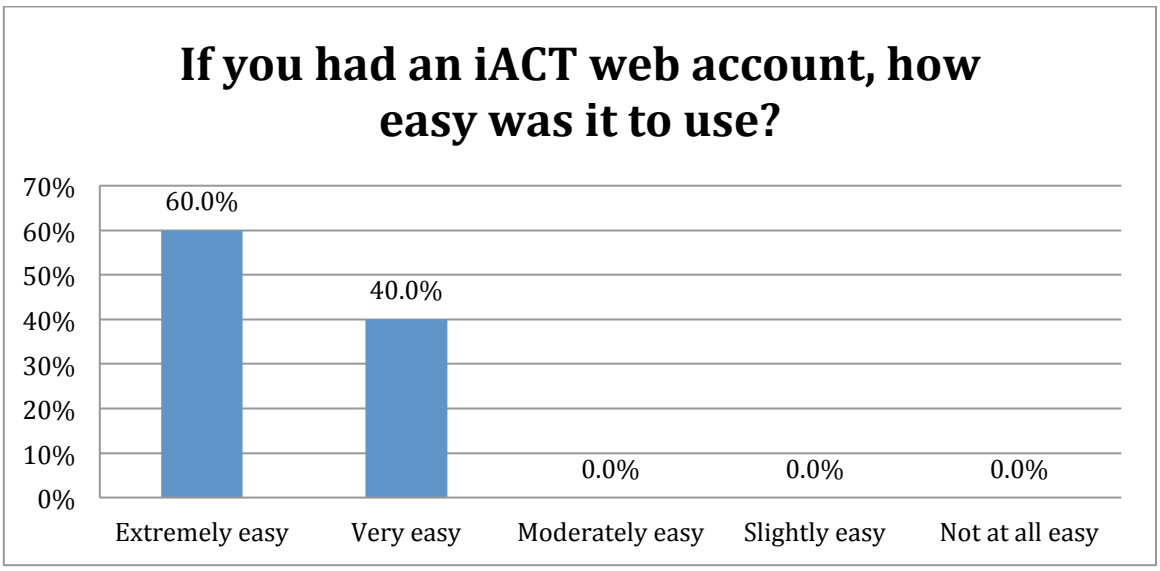



I like the look and feel of the application.
60%
50%
40%
30%
20%
10%
0%
57.1%
14.3%
28.6%
0.0%
0.0%
Strongly agree
Agree
Neutral
Disagree
Strongly disagree


Were you able to add a new thought?
100%
50%
0%
100.0%
0.0%
0.0%
Yes
No
Unaware of feature


Were you aware thoughts could be deleted?
100%
80%
60%
40%
20%
0%
85.7%
14.3%
0.0%
Yes
No
Unaware of feature





[64] Parse, 2012. *The mobile app platform for developers*. [Online] Available at: https://www.parse.com [Accessed 10 June 2012].

6.Conclusions and Future Work

This chapter provides the conclusions I have drawn from iACT's implementation as well as an analysis of future work that could be carried out. The finished project could be improved and extended in a number of significant directions. Many of these improvements are refinements to existing functionality that would help provide a better ACT based experience for the iPhone application's users. I also discuss the work that would be required to take iACT to a trial in a real therapy environment.

6.1 Conclusions

The primary goal of this project was to produce a software system that could augment an ACT process. To achieve this, software with several key 'must have' and 'should have' objectives was specified as shown in *figure 34.* This project has successfully implemented each of the features shown, with the sole exception of random ACT messages.

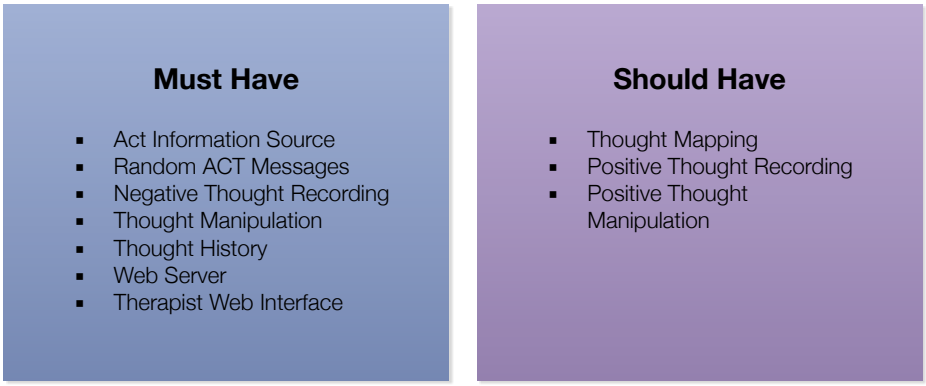


Figure 34 'Must Have' and 'Should Have' Objectives

6.1.1 Main Achievements

The completion of this project has resulted in the following main achievements:

- Successful development and implementation of a mobile system for ACT based thought recording and manipulation.
- Successful development of an API and server system to capture and transmit data recorded by the mobile system.
- Successful deployment of a web application to facilitate therapist access to data recorded by the mobile system.
- Successfully satisfied almost all of the client's specifications.

Within each of these have been a large number of implementation objectives that have been completed to meet the client's goals for the system.

6.1.2 Implementation

The implementation of this project saw the creation of three key components, the iPhone application, REST API and web application. These were delivered through a substantial software system, comprising over eight thousand lines of code. The following is an overview of the features I have implemented for each of these key components.

- Mobile system for ACT based thought recording and manipulation

This was achieved by iACT's iPhone application. As per the client's goals, it provides a number of features to satisfy the requirement objectives:

- Information on ACT and the ACT model is provided by means of an interactive ACT 'hexaflex'. As the client requested, tapping elements of the hexaflex provides further information about that ACT principle.
- Thought recording capability. This allows the user to quickly record and rate positive or negative thoughts. These thoughts are logged in the application and submitted to a web server.
- Users are able to interact with recorded thoughts to aid in ACT's cognitive diffusion techniques. However, as I noted in the evaluation, this feature could use further development to deliver measurable therapeutic benefits.
- Thought locations are logged using the iPhone's GPS feature. This data is then used to populate maps allowing the user to observe any geographic trends in their thought's occurrences.

The iPhone application is one of the project's main successes. I was able to utilise the latest in best practices for iOS development, moving to an ARC code base and using the latest storyboard techniques to develop a complex design. The application uses a user interface paradigm that is common to a huge number of iOS applications, resulting in software that is easy to use. Common tasks can be achieved in a minimum number of interactions. The application also stores data using a Core Data based model. Core Data is one of the most powerful and complex frameworks present in iOS and delivers a data model that is powerful, fast, and easily extended to incorporate new features. The application also uses many other advanced iOS features, such as Core Location, MapKit and table views.

I was also able to add functionality beyond just the core 'must have' functionality. The software additionally has some advanced mapping features. With each thought recorded, the phone's GPS feature is used to capture its latitude and longitude. This information is then used to generate maps showing the locations of the thoughts.

The application has also been successful in incorporating features of ACT. It helpfully provides the user with information on ACT and its core principles using an interactive hexaflex that the user may be familiar with from their own therapy sessions. The design of iACT also helps enforce some of these core principles. For example, by recording the thoughts the user can exercise the 'mindfulness' and 'acceptance' skills that ACT uses to deal with negative thoughts. Therapists can also use the software to send new thoughts to service

user's handsets should the course of discussions during a session lead them to believe there is a thought that has gone unnoticed that should be recorded.

The iPhone application also hoped to deliver features that aid in ACT's cognitive diffusion techniques. As I noted in the evaluation, this is the aspect of the software that could most benefit from more development time. Whilst I have implemented a system to permit the thought to be manipulated on the screen, it still remains at all times a string of text. It would have been nice to have created interactions that further abstract the on screen representation of the thought, perhaps with the letters re-arranging themselves or transforming. This would provide further therapeutic benefits.

- API and server system for data capture and transmission

This was achieved by iACT's REST API. This provides features that help deliver the goals of the iPhone application and web interface:

- Simple REST based design permitting interactions between the server and the iPhone.
- JSON based data format for open exchange with any software or web based service.
- Ruby on Rails based server capturing iPhone application data and allowing therapists to transmit new thoughts to their client's handsets.

The API used by the system has resulted in a fast, extensible and easy to use method to interact with iACT's web server. Entirely HTTP based and similar to REST, it can be incorporated easily into other web sites or applications. Its use of JSON results in easy to understand data with small file sizes, ensuring good performance on poor internet connections. Whilst the present API is perfectly functional, with more time it could be refined to conform fully to REST design principles.

- Web application for therapist users

This was achieved by iACT's Ruby on Rails web server. This facilitated a web application to provide the following features:

- Web interface allowing therapists access to thought data.
- Ability to control registration of new therapist and iPhone application user accounts.
- Ability to send new thoughts to service users.

The finished web application took advantage of the Rails and Bootstrap frameworks to deliver a web site with an easy to use interface. Its model view controller based architecture makes it easy to extend or add more functionality, meaning that the types of data and the way in which it is presented can be developed further in the future. The design also conforms to modern web standards using mainstream technologies, ensuring compatibility with a wide range of browsers and devices. I was also able to develop the site beyond its original requirements, with some attention given to developing an appropriate security model to control access to the sensitive data it stores.

6.1.3 Testing and Evaluation

To ensure delivery of a high quality finished system I adopted several different testing and evaluation strategies. For the iACT web site I used a 'test-first' methodology that meant I was able to design, implement and test the web site simultaneously. This approach allowed for rapid implementation of features while maintaining a high standard of software quality. It also left me with a full suite of unit tests that could be continually repeated, ensuring that new features did not break existing functionality.

To test the iACT iPhone application a 'black box' testing methodology was chosen. As I was both learning how to program for the iOS and implementing the application at the same time I chose not to use a unit testing system. This is because the implementation of the application changed dramatically, potentially rendering any unit testing system broken repeatedly. Rather than constantly redesign unit tests I elected to

use the black box approach. With each iteration every feature and function of the application would be checked again with issues fixed or debugged as they arose.

It had been agreed during the proposal period that iACT would not be trialed with real users or submitted to the iOS app store at this stage. This meant that the evaluation of iACT would primarily occur by comparing the finished product against the client's original needs. To this end I reappraised my work from the perspective of the software requirements. The final element of my evaluation was a small degree of user testing. Whilst these users are not representative of iACT's intended user base, performing a small degree of research into their experiences with the software was still helpful. The results of these user's experiences allowed me to ensure that other people could understand how to use the software, and give feedback on its design.

6.1.4 Reflections

My personal aim throughout this process was to deliver software that met or exceeded the client's needs. I also wanted a system whose quality was comparable to the existing software products I had tried during the proposal and background research phases. Finally, I wanted to deliver the system in a way that permitted easy future developments using best practices and the latest technologies. Looking back at the development of this system, it is undoubtedly the extent of its complexity that has been most striking. During the proposal phase I did not foresee how time consuming 'simple' features such as an effective data structure for the iOS application would become. Nor did I expect the design and implementation of an API for the iOS application to interact with the web site to be so complex. Despite this, I feel I have achieved my personal aims during iACT's development.

One of the primary sources of this project's challenges has undoubtedly been its tiered architecture. Because the iACT system reaches across several different computer systems, with different programming languages and development environments for each, understanding how the various pieces interact with one another was highly complex. When issues arise with tiered architectures one cannot simply rely on a debugger or terminal output to provide an explanation of what has gone wrong. The developer needs to have a deep understanding of what is happening on each part of the system as the user performs interactions to understand where to look for the solutions to bugs.

With the different tiers of the system each running their own native code, the exchange of data required complex conversions to open formats for communication purposes. In iACT's case JSON formed the 'glue' between each layer, allowing them to connect with one another. Ensuring that each tier produced valid JSON and then parsed this into its own native implementation without corrupting or changing the data was another major challenge.

Another of the challenging aspects of this implementation has been the complexity of the iPhone and web application's development environments. Both Xcode and Ruby on Rails are enormously powerful and learning even a partial extent of the many libraries and features they offer the developer was very time consuming. This is before one even considers the enormous range of third party extras available for both.

The finished system compares favourably to the existing software products I examined during my background research, Bipol and Poo Review. iACT provides an interface that conforms much more to Apple's official Human Interface Guidelines (HIG)[65] than Bipol. This is largely due to Bipol's heavy reliance on difficult to understand custom user interface elements. One of the guiding principles in the HIG is that "People Expect to Find iOS Technologies in the Apps They Use."[66] This principle is one I have followed closely, using only commonly found user interface elements and controls to make the application as easy to use as possible.

As I mentioned in the design and implantation chapters, one of my goals for this project was to adopt industry standards and best practices where possible in order to gain the greatest personal benefit from this project. It is in this regard I feel the system to be most successful. I have not taken any easy options or short cuts in any

of the code I have written. With the knowledge I gained during the development process I would undoubtedly now be at a great advantage should I begin development of another iOS project.

6.2 Future Work

There are a number of directions in which iACT could be further developed. This section outlines these, and where possible provides some detail as to how this work might be carried out. Whilst the project has delivered a fully functional system, there are several ways in which it could be changed or developed to provide a better experience for its users.

6.2.1 Push Notifications

One of the most important developments to add to the system would be a push notification system for sending ACT related messages. When I wrote the project proposal it was written largely in ignorance of the work levels involved to deliver each feature of the system. This was inevitable given my relative lack of experience. I was quickly to discover that push notifications are a complex feature to implement. In the time available to deliver the project I was sadly unable to deliver this feature. However, I do have a good understanding of how such a feature would work and, given more time, it is certainly something I would have been able to add to the system.

My Initial assumptions regarding push notifications had been that it would be possible to send a notification simultaneously to all users of the iACT iOS application in one single push from the iACT server. Upon exploring Apple's API[67] for push notifications I learned that this is not the case. Instead one must push each notification individually to each individual iPhone handset, not the application itself. This is achieved by obtaining a unique identifier from each handset and registering it against a user in a database. One then must iterate through all of the users sending a message to each handset individually. This would therefore require the creation of a database system to manage the adding and removing of unique identifier keys, along with code in the iACT iOS application to submit the user's key to the server.

When performing the submission, a JSON file must be sent to Apple's push notification servers. This must contain the unique ID and the content of the message to be sent in a specific format. This must also be sent along with a special certificate key that must be created to confirm you have a valid iOS developer account. All of this would have required the construction of a fairly complex database system to manage the keys, and to spend time building this would have almost certainly have been to the detriment of the rest of the system. I therefore unfortunately had to take the decision to drop this feature for the time being.

Given more time, I suspect I would have modified the service user table in the iACT Online website database to include an additional attribute to contain the user's handset unique ID. An extra database entity would have been added to store the content of the random messages to send to iACT's users. A cron[68] job would then be established on the iACT online server to send at random intervals a message randomly chosen from the iACT messages table. The existing Rails based server would in all likelihood be capable of modification to support this.

6.2.2 Refined API

While my API is fully functional to the point where I could hand another developer the documentation and they would be able to use it for all of the necessary functionality, I do feel it could be refined further. In particular, I would like for the API to be fully compliant with the definition of a RESTful API. This would make it even easier for developers with prior experience in REST architectures to use. Ruby on Rails is certainly able to deliver such an API. I would also of course have to finish implementing a security scheme for the API. In all likelihood this would have to eventually be built around some kind of 'shared secret' style architecture in which an API security key would be sent to the requesting service upon first login attempt. This is the approach adopted by the Foursquare and Twitter REST APIs and would prove effective here.

6.2.3 Search Functions

Although search functions were not explicitly a feature to be found in the proposed requirements, search boxes were implemented in my wireframes and mock user interfaces for both the web and iOS applications. This was another feature that would have been implemented given more time. As both the site and the iOS application use an SQLite 3 database to underpin their data models it should be possible to write efficient search algorithms to return results quickly.

For the website search would implemented in three locations. Firstly a search field would be added to the therapist list page so that administrative users can quickly find a therapist for modification or removal. A search field would also be present on the service users listing page. Again, this would be to permit quick retrieval of a specific service user. Lastly a search box would also be implemented on the service user's thought listing page.

In the iOS application search would be found solely on the thought history page. In my mock interfaces I designed a user interface similar to the one found in the iPhone's built in music player application. At the top of the list of thoughts would be a search bar. Typing a search query here would cause the list of thoughts to filter accordingly.

6.2.4 Advanced Thought Interactions

The presently implemented thought interaction is quite basic. At this stage I simply wanted to demonstrate the concept and give the client and the users a strong indication of how the application would work. One of the core goals of ACT is to change the context in which a thought is understood to gain distance from the content of harmful thoughts. This will ultimately require some advanced graphical manipulation to animate the recorded thought in response to input by the user. In order the prove that a mobile application can become a serious ACT tool with the capacity to demonstrate a real improvement in peoples perception of their own thoughts this feature will require a great deal more work. Given the constraints on my time, there simply wasn't scope to learn complex animation code to deliver the quality of interactions I would have liked.

A good example of the sort of improved interaction that could be created would be to dynamically rearrange the words entered. As ACT is keen to change the context of thoughts one idea that I raised with the client was to incorporate some kind of anagram solving code. This code could then change the meaning of the sentence entirely, doing so at random. This is was an idea to which the client was extremely positive, and is certainly something to consider for the future.

6.2.5 Moving to a Production Environment

To proceed to a real life production environment with my system will require some more work. As it was never a goal of the development process to submit the application to the App Store, I have not considered in full the extensive list of requirements that have to be satisfied to pass Apple's screening process[69]. There are doubtless a few loose ends that would require a small amount of additional code to ensure the application is fully compliant.

The application could also use some kind of terms and conditions screen explaining in full how the service works. Users will rightly expect to be fully informed about what exactly the data is collected for, and specifically what data is sent to their therapist. This would allow users to make an informed decision as to whether or not they are comfortable in using the software.

The last significant change would be a detailed and well-researched approach to the system's security. Perhaps more than ever before, software users are extremely conscious of the security of their data online. This is in large due to the number of high profile data loss incidents in recent years. As the development process was never intended to yield a final production ready version of the software this was never included in any great detail in the proposal. I have where possible tried to implement a security system, but as I

discussed in the implementation chapter, at present the API for submitting and receiving thought data is insecure. Securing HTTP communications is a complex problem given HTTP's stateless nature and addressing this fully would require careful thought, design and research.

---

[65] Apple, 2012. *iOS Human Interface Guidelines*. [Online] Available at: http://developer.apple.com/library/ios/#DOCUMENTATION/UserExperience/Conceptual/MobileHIG/Introduction/Introduction.html [Accessed 15 August 2012].

[66] Apple, 2012. *iOS Human Interface Guidelines*. [Online] Available at: http://developer.apple.com/library/ios/#DOCUMENTATION/UserExperience/Conceptual/MobileHIG/Introduction/Introduction.html [Accessed 15 August 2012].

[67] Apple, 2012. *Apple Push Notification Service*. [Online] Available at: http://developer.apple.com/library/ios/#documentation/NetworkingInternet/Conceptual/RemoteNotificationsPG/ApplePushService/ApplePushService.html#//apple_ref/doc/uid/TP40008194-CH100-SW9 [Accessed 20 June 2012].

[68] Wikipedia, 2012. *cron*. [Online] Available at: http://en.wikipedia.org/wiki/Cron [Accessed 02 August 2012].

[69] Apple, 2012. *App Store Review Guidelines*. [Online] Available at: https://developer.apple.com/appstore/resources/approval/guidelines.html [Accessed 17 August 2012].

7. Bibliography

Apple, 2011. *What's New in Xcode 4*. [Online] Available at: https://developer.apple.com/technologies/tools/whats-new.html#llvm-compiler [Accessed 17 June 2012].

Apple, 2012. *App Store Review Guidelines*. [Online] Available at: https://developer.apple.com/appstore/resources/approval/guidelines.html [Accessed 17 August 2012].

Apple, 2012. *Apple Push Notification Service*. [Online] Available at: http://developer.apple.com/library/ios/#documentation/NetworkingInternet/Conceptual/RemoteNotificationsPG/ApplePushService/ApplePushService.html#//apple_ref/doc/uid/TP40008194-CH100-SW9 [Accessed 20 June 2012].

Apple, 2012. *iOS 6 Preview*. [Online] Available at: http://www.apple.com/ios/ios6/ [Accessed 19 May 2012].

Apple, 2012. *iOS Developer Program License Agreement*. [Online] Available at: https://developer.apple.com/programs/terms/ios/standard/ios_program_standard_agreement_20120611.pdf [Accessed 30 May 2012]. Requires an Apple developer account to view.

Apple, 2012. *iOS Human Interface Guidelines*. [Online] Available at: http://developer.apple.com/library/ios/#DOCUMENTATION/UserExperience/Conceptual/MobileHIG/Introduction/Introduction.html [Accessed 15 August 2012].

Apple, 2012. *WWDC 2012 Keynote Presentation*. [Online] Available at: http://www.apple.com/apple-events/june-2012/ [Accessed 14 June 2012].

Apple, 2012. *Xcode 4*. [Online] Available at: https://developer.apple.com/xcode/ [Accessed 28 May 2012].

Bare Bones Software, 2012. *BBEdit 10*. [Online] Available at: http://www.barebones.com/products/bbedit/ [Accessed 20 May 2012].

bcrypt-ruby, 2012. *bcrypt-ruby*. [Online] Available at: http://bcrypt-ruby.rubyforge.org [Accessed 08 June 2012].

Beating Bipolar, 2012. *Bipol-App*. [Online] Available at: https://www.beatingbipolar.org/bipol-app/ [Accessed 16 February 2012].

Bohlmeijer, E. T., Fledderus, M., Rokx, T. A. J. J., & Pieterse, M. E., 2011. Efficacy of an early intervention based on acceptance and commitment therapy for adults with depressive symptomatology: Evaluation in a randomized controlled trial. *Behaviour Research and Therapy*, 49(1), p.62.

Bookwalter, J.R., 2010. *Apple Facing Federal Probes Over Section 3.3.1 of iPhone SDK?* [Online] Available at: http://www.maclife.com/article/news/apple_facing_federal_probes_over_section_331_iphone_sdk [Accessed 29 May 2012].

Brichter, L., 2010. *US Patent & Trademark Office*. [Online] Available at: http://appft.uspto.gov/netacgi/nph-Parser?Sect1=PTO2&Sect2=HITOFF&u=%2Fnetahtml%2FPTO%2Fsearch-adv.html&r=1&f=G&l=50&d=PG01&p=1&S1=20100199180.PGNR.&OS=dn/20100199180&RS=DN/20100199180 [Accessed 19 August 2012].

Brownlee, J., 2010. *Twitter Vows Never To Weaponize Their Patents, Including Pull-To-Refresh*. [Online] Available at: http://www.cultofmac.com/161367/twitter-vows-never-to-weaponize-their-patents-including-pull-to-refresh/ [Accessed 19 August 2012].

Butler, E., 2010. *Firesheep*. [Online] Available at: http://codebutler.com/firesheep [Accessed 19 June 2012].

Callahan, K., 2012. *Accessorizer*. [Online] Available at: http://www.kevincallahan.org/software/accessorizer.html [Accessed 29 June 2012].

Coia, G., 2012. *github*. [Online] Available at: https://github.com/giobox [Accessed 20 May 2012].

Crockford, D., 2006. *The application/json Media Type for JavaScript Object Notation (JSON)*. [Online] JSON.org Available at: http://www.ietf.org/rfc/rfc4627.txt [Accessed 30 May 2012].

Django, 2012. *The Web framework for perfectionists with deadlines*. [Online] Available at: https://www.djangoproject.com [Accessed 27 May 2012].

Doherty, G., Coyle, D., & Matthews, M., 2010. Design and evaluation guidelines for mental health technologies. *Interacting with Computers*, 22(4), p.243.

Doxygen, 2012. *Doxygen*. [Online] Available at: http://www.stack.nl/~dimitri/doxygen/ [Accessed 05 August 2012].

Dropbox, 2012. *Dropbox*. [Online] Available at: https://www.dropbox.com [Accessed 20 May 2012].

Fielding, R., 2000. *Architectural Styles and the Design of Network-based Software Architectures*. [Online] Available at: http://www.ics.uci.edu/~fielding/pubs/dissertation/top.htm [Accessed 05 June 2012].

Fitzpatrick, J., 2010. *Firesheep Sniffs Out Facebook and Other User Credentials on Wi-Fi Hotspots*. [Online] Available at: http://lifehacker.com/5672313/sniff-out-user-credentials-at-wi+fi-hotspots-with-firesheep [Accessed 18 June 2012].

Flask, 2012. *Flask, Web development one drop at a time*. [Online] Available at: http://flask.pocoo.org/docs/ [Accessed 28 May 2012].

foursquare, 2012. *foursquare*. [Online] Available at: http://www.foursquare.com [Accessed 03 June 2012].

Git, 2012. *Git Version Control*. [Online] Available at: http://git-scm.com [Accessed 01 June 2012].

Harris, R., 2006. Embracing Your Demons: an Overview of Acceptance and Commitment Therapy. *Psychotherapy in Australia*, 12(4), p.2.

Hartl, M., 2012. *Ruby on Rails Tutorial: Learn Web Development with Rails*. [Online] Available at: http://ruby.railstutorial.org [Accessed 01 June 2012].

Joyent, 2012. *Node.js*. [Online] Available at: http://nodejs.org [Accessed 28 May 2012].

Lathia, N., 2011. *Using Idle Moments to Record Your Health via Mobile Applications*. Computer Laboratory, University of Cambridge.

Lathia, N., 2012. *The (Poo) Review*. [Online] Available at: https://play.google.com/store/apps/details?id=com.poo.review&hl=en [Accessed 20 May 2012].

legislation.gov.uk, 1998. *Data Protection Act*. [Online] Available at: http://www.legislation.gov.uk/ukpga/1998/29/contents [Accessed 14 June 2012].

Mark, D., Nutting, J. & LaMarche, J., 2011. *Beginning iOS 5 Development: Exploring the iOS SDK*. 1st ed. London: Apress.

Milowski, A., 2012. *poster-extension*. [Online] Available at: http://code.google.com/p/poster-extension/ [Accessed 20 June 2012].

mislav, 2012. *will_paginate*. [Online] Available at: https://github.com/mislav/will_paginate/wiki/ [Accessed 19 June 2012].

OAuth, 2012. *OAuth*. [Online] Available at: http://oauth.net [Accessed 11 June 2012].

Office for National Statistics, 2011. *Internet Access - Households and Individuals*. [Online] Available at: http://www.ons.gov.uk/ons/dcp171778_227158.pdf [Accessed 06 February 2012].

Omni Group, 2012. *OmniFocus for Mac*. [Online] Available at: http://www.omnigroup.com/products/omnifocus/ [Accessed 20 May 2012].

Parse, 2012. *The mobile app platform for developers*. [Online] Available at: https://www.parse.com [Accessed 10 June 2012].

Rails, 2012. *Ruby on Rails*. [Online] Available at: http://rubyonrails.org [Accessed 28 May 2012].

Ray, J., 2012. *Sam's Teach Yourself iOS 5 Application Development in 24 Hours*. 3rd ed. Sams.

RestKit, 2012. *RestKit*. [Online] Available at: http://restkit.org [Accessed 29 May 2012].

Rodriguez, A., 2008. *RESTful Web services: The basics*. [Online] Available at: https://www.ibm.com/developerworks/webservices/library/ws-restful/ [Accessed 02 August 2012].

RSpec, 2012. *RSpec*. [Online] Available at: http://rspec.info [Accessed 05 June 2012].

Schimmel-Bristow, A., Bricker, J. B., & Comstock, B., 2012. Can Acceptance & Commitment Therapy be delivered with fidelity as a brief telephone-intervention? *Addictive Behaviours*, 37(4), p.517.

Siracusa, J., 2011. *Mac OS X 10.7 Lion: the Ars Technica review*. [Online] Available at: http://arstechnica.com/apple/2011/07/mac-os-x-10-7/10/ [Accessed 17 June 2012].

SQLite Database Browser, 2012. *SQLite Database Browser*. [Online] Available at: http://sqlitebrowser.sourceforge.net [Accessed 10 June 2012].

Stanford University, 2011. *iPad and iPhone Application Development*. [Online] Available at: http://itunes.apple.com/gb/course/developing-apps-for-ios-fall/id495054839 [Accessed 01 February 2012].

Twitter, 2012. *Bootstrap*. [Online] Available at: http://twitter.github.com/bootstrap/ [Accessed 10 June 2012].

twitter, 2012. *twitter*. [Online] Available at: http://www.twitter.com [Accessed 04 June 2012].

W3C, 2000. *Simple Object Access Protocol (SOAP) 1.1*. [Online] Available at: http://www.w3.org/TR/2000/NOTE-SOAP-20000508/ [Accessed 06 May 2012].

Wikipedia, 2012. *cron*. [Online] Available at: http://en.wikipedia.org/wiki/Cron [Accessed 02 August 2012].

Wikipedia, 2012. *Gold plating (software engineering)*. [Online] Available at: http://en.wikipedia.org/wiki/Gold_plating_(software_engineering) [Accessed 19 August 2012].

Wikipedia, 2012. *Object-relational mapping*. [Online] Available at: http://en.wikipedia.org/wiki/Object-relational_mapping [Accessed 03 June 2012].

Wikipedia, 2012. *Serialization*. [Online] Available at: http://en.wikipedia.org/wiki/Serialization [Accessed 09 June 2012].

Williams, J. M. G., Barnhofer, T., Crane, C., Herman, D., Raes, F., Watkins, E., & Dalgleish, T., 2007. Autobiographical memory specificity and emotional disorder. *Psychological Bulletin*, 133(1), p.122.

Xamarin, 2012. *MonoTouch*. [Online] Available at: http://xamarin.com/monotouch/ [Accessed 28 May 2012].

YARD, 2012. *Yay! A Ruby Documentation Tool*. [Online] Available at: http://yardoc.org [Accessed 11 August 2012].