CHALMERS

Pomodoro Project

Pomodoro Project for iPhone

An evaluation of the iPhone platform and a description of techniques for iPhone development.

Master of Science Thesis in the Programme Interaction Design

RASMUS WAHLMAN
MICHAEL SANDÉN

Department of Computer Science and Engineering
CHALMERS UNIVERSITY OF TECHNOLOGY
UNIVERSITY OF GOTHENBURG
Göteborg, Sweden, June 2010

Rasmus Wahlman
Micahel Sandén

Examiner: Sven-Arne Andreasson

Department of Computer Science and Engineering
Chalmers University of Technology
SE-412 96 Göteborg
Sweden
Telephone + 46 (0)31-772 1000

Cover:
The logo for the Pomodoro Project.

Department of Computer Science and Engineering
Göteborg, Sweden June 2010

Abstract

Mobile phones today are used for much more than just calling and text messaging people. As technology has become more advanced, developers have started to incorporate a wide variety of functionality into their phones, such as the ability to surf the web, listen to music and also run application on their phones. This report will deal with the evaluation of such a phone, namely the iPhone. This report will also deal with the development of an iPhone application, which is a digital version of the Pomodoro Technique, with large focus on usability and interaction design.

The first part of the project constitutes for a pre-study part within the area of iPhone development as a whole with information about general iPhone development tools, frameworks, design and interface design patterns. Since there are many ways to implement a product which must have high user-friendliness and usability, we will therefore explain and discuss the process behind developing the Pomodoro Application and motivate our choices of design methods and general approach towards development.

By working on this project, we have realised that developing applications for the iPhone is rather simple as long as the developer sticks to standard frameworks and interface patterns supported by the iPhone SDK itself. If you wish to create an interface of your own however, you will be forced to do it all by yourself and it will be much more time consuming in the end.

Sammanfattning

Mobiltelefoner används idag till mycket mer än bara telefonsamtal och textmeddelanden. I takt med att tekniken blir mer avancerad har företag börjat lägga in allt mer funktionalitet i sina telefoner, så som möjligheten att surfa på nätet, lyssna på musik och köra applikationer på sina telefoner. Denna rapport kommer behandla en evaluering av en sådan telefon, nämligen Apples iPhone. Rapporten kommer även behandla utvecklingen av en applikation till iPhone som är en digital version av Pomodoro-tekniken med stort fokus på användarvänlighet.

Den första delen av projektet bestod utav en förstudie vars syfte var att få en överblick över iPhone-utveckling samt vilka ramverk och verktyg som finns att tillgå. Eftersom det finns många sätt att implementera och designa en applikation kommer rapporten även ta upp de metoder som används under utvecklingen av applikationen samt designmetoder för att ta fram ett användarvänligt gränssnitt.

Det vi kommit fram till genom att göra det här projektet är att det är relativt enkelt att utveckla applikationer till iPhone så länge som man håller sig till standardramverk och vanliga interfacedesignmönster som används till iPhone. Om man däremot vill skapa ett användargränssnitt som är helt eget får man räkna med att få göra allt själv och att utvecklingen kommer ta mycket längre tid.

Table of Contents

1. Introduction

Mobile phones have existed in the mainstream market since the mid-nineties. During this time, mobile phones have evolved from being simple voice call phones into phones with functionality reminiscent of palm computers. Today, people do not use their phones just to make calls and send SMS messages, people use their phones to listen to MP3s, surf the web, play games and run many other types of applications. In recent years, Apple has released their *iPhone* with its operating system *iPhone OS* and Google has released their mobile operating system *Android* together with APIs allowing users to make their own applications for these platforms. This together with the success of online application stores such as Apple's own *iPhone App Store* makes the iPhone and Android phones attractive markets for companies and hobby programmers alike.

This report will deal with the planned development of a project titled *Pomodoro Project* for iPhone as an assignment given by Logica. The project is carried out by teams of students as their master thesis work and the goal of the project is to evaluate the technology behind various mobile platforms, including the iPhone and Android phones. To evaluate these platforms, each team will investigate technologies associated with each platform in order to identify possibilities and limitations of these platforms. Based on the evaluation, each team will implement an application to prove the results of their research. The application is a tool helping the user work according to the *Pomodoro Technique* (2009). The fundamental idea of the Pomodoro technique is to work as efficiently as possible during limited time slots called *Pomodoros* and take notes of any interruptions that may happen. Apart from the evaluation that every team will perform, we will also have our main focus on interaction design. This means that we will put effort into investigating design patterns and develop our user interface according to best practices from interaction design methodology.

The report will also deal with various aspects of iPhone development and the technologies/frameworks behind it and should be seen as a short introductory guide to iPhone development.

1.1 Logica

Logica (formerly known as WM-Data) is an international IT-consultant company employing roughly 40,000 employees all over the globe. Logica is currently looking into researching new mobile technologies (iPhone, Android etc) in order to evaluate the possibilities and benefits of these new platforms.

2. Method

The first objective of the project is to evaluate technologies associated with the iPhone. We will gather information and study the possibilities to achieve the best result for the Pomodoro project. We will summarize the necessary findings to provide the reader with basic knowledge about some of the technologies that exist as well as preparing them for development with the iPhone SDK.

We will continue with a study of our key focus in the thesis which mainly consists of interface design practices on the iPhone. However, we will also provide a small comparison between iPhone and Android platforms. To succeed with our key focus, we must start to familiarize ourselves with the iPhone and its common interface patterns/components. Therefore, studying iPhone applications is something we must do in order to draw any conclusions of how the interface design patterns are used to structure an application.

The next step will be to search for any complementary information that Apple might provide about interface design as well as how to use their interface components on the iPhone.

When we feel confident enough to start designing the interface, we will draw a skeleton interface design using pen and paper. We prefer to design on paper before implementing anything since it is a really quick way to get started. This will in turn help us receive early feedback about the design and we can also use the papers to perform user tests. In some stage before implementing anything, we will perform user tests using the design method *paper prototyping*. Basically, the purpose behind the method is to let users test the application as if it was the real application even though it is completely constructed out of papers. Hopefully, it will discover possible flaws in the design already at an early stage preventing us from changing the interface in the future.

We will perform proof of concept of the necessary functionality to be integrated in the Pomodoro Project. However, we won't implement anything of the reference project until we finished the interface as well as the user testing. The design should be completed before adding the actual functionality in the back-end. This to avoid changes in every part of the code and structure.
Once that we can start with the implementation of the reference project we will start building the interface using the interface builder that comes with the iPhone SDK. Then we will continue implementing the functionality for the Pomodoro technique in a prioritized order. This to make sure that the most essential functionality is added first.

3. Pre-study

To implement the Pomodoro Application, we must perform research in order to identify ways of implementing various parts of it. This includes finding frameworks for various purposes, development tools, design patterns and also investigate aspects such as security and maintainability among other things. The following chapter contains the results of our research.

3.1 Development Tools

This section will deal with different tools and frameworks that could be useful for iPhone development and there is also a discussion about the advantages and disadvantages of making a native application and a web application solution.

3.1.1 Cocoa

Cocoa is a set of object-oriented frameworks that provide a runtime environment for applications running on Mac OS X and iPhone OS. It is also part of a development environment that makes it possible to build applications for the two platforms Mac OS X operating system and iPhone OS, the operating system for multi-touch devices such as iPhone, iPod touch and iPad. However, Cocoa is the the only application environment for iPhone OS and it supports only application development and not development of any other kind of executable on the iPhone OS. This means that the developer can't develop software close to the operating system or close to the hardware.

3.1.2 Objective-C

Objective-C is the main supported language on the iPhone, although there is support for other languages through various third party frameworks. Objective-C is a superset of regular C, meaning that it is simply an extended version of C with more functionality and additional syntax, although it is possible to mix C and Objective-C syntax seamlessly.

One of the major differences between C and Objective-C is that the latter has support for some object-oriented features such as classes and inheritance (although only single inheritance is supported). Another Objective-C specific feature is a message passing syntax that is used for executing methods on objects and accessing instance variables among other things. The advantage of using this type of message passing over regular function calls is the fact that it is possible for an object to forward the message elsewhere if the object itself is not capable of resolving the message it received. Finally, Objective-C has some additional memory management functionality that is used for handling memory occupied by objects, which is described in a later section.

3.1.3 iPhone SDK

The iPhone SDK contains the most commonly used frameworks and tools for iPhone development. It has frameworks for interface design, network communication, database communication, audio, graphics and much more. For more details, check the documentation that can be found on iPhone Developer Connection.

3.1.3.1 XCode

XCode is the main IDE used for iPhone development and comes with the iPhone SDK. Similarly to many IDE's such as Visual Studio, the interface is divided into a code view and tree view that organises code files and other resources. XCode also has a debugging tool similar to many other IDE's, but this will be dealt with in a later section.

The XCode IDE also contains a tool called *Interface Builder*. This tool allows you to create interfaces by drag-and-dropping common interface components onto a preview window representing the current state of the interface. It is also possible to specify methods that respond to certain events associated with interface components and to some extent change the look of various interface components. Interfaces can of course be written through code, though this can probably be more time consuming.

3.1.4 iPhone Native Apps Vs. iPhone Web Apps

One question one may ask oneself is: should the application be developed as a native application or as a web application? An iPhone web application is a web site that is optimized to be compatible for viewing on the iPhone, but how much can a well executed web app do compared to a native application? Apart from the fact that it does not work off-line, there are a lot of aspects to consider such as what is the purpose of the application and performance related requirements.

3.1.4.1 Native iPhone Application

If the purpose of the application is to take advantage of features built into the iPhone itself such as motion detection, voice detection, camera and GPS then it is necessary to make it a native application. As of now, it is only possible to access these phone features via offline frameworks.

As mentioned above, one reason to make it a native application is to assure that the content or service is available offline. Two arguments to support a native application in this case is, if the main purpose of the application is to make the content as quickly and easily available as possible, or without the dependability of being connected to a steady internet connection. Native applications are meant to be faster and to improve the performance of an offline application, it is possible to define your own caching strategies, your own network services and your own event/threading models.

The decision of whether to make a native application or not is also dependent on how you want to distribute your application and if you want to make a profit from it. If you want to make money from selling the application itself, then the choice is definitely to make a native application. The app store provides an easy way for developers to distribute their applications, however, Apple must approve of your application before it is posted in the app store. Of course, there is no guarantee that putting an application on the app store will automatically lead to profit, on the contrary, many of the applications in the app store remain unnoticed.

3.1.4.2 Web iPhone Application

One reason to make a web app is if you have a web site and you want a similar interface and content to feature on your iPhone application. It is possible to use the same publishing system as for your main web site since essentially the only difference between a web site and an iPhone web application is that is optimized for viewing in the iPhone.

Another good reason for making a web application is the fact that Apple has no approval process for web applications. If you believe that the content of your application is something that Apple could call inappropriate or you know that it has a chance of getting rejected, you should consider making your application web based.

Furthermore, if you are interested in targeting multiple platforms, then it might be a good idea to make your application web based. Building the web application for iPhone doesn't necessary mean that it can't be compatible on different platforms. iPhone web applications are displayed in Safari, which uses the open source framework Webkit. Since Webkit is also the foundation of other browsers on other platforms such as Android and some Blackberry devices, it is possible to port over the web application with different levels of effort depending on the complexity of the web application.

The disadvantage of making a web-based application is the fact that you can not make a profit from it via the app store. Publishing your application on the app store also has the advantage of getting your application published in a commonly visited place, making it likely that some people will notice your application. On the other hand, by not publishing your application on the app store, Apple can not seize 30% of the profit you make on sold applications. However, according to Brown (2009), vice president of media and entertainment and head of mobile solutions practice at Digitaria, web applications have no proven models for paid access, and iTunes makes it way too easy for users to spend their money on native applications.

Finally, these are merely an insight of aspects that should be taken into consideration before deciding if the application should be of native or web.

3.1.5 Design patterns

Design patterns are program structures used to solve common problems, separate concerns and improve understandability of code among other things.

Common design patterns used on the iPhone are:

- Delegation
- Model View Controller
- Target-Action

Read about Cocoa Design Patterns for more information (Apple Inc., 2009).

3.1.5.1 Delegation

Delegation is a pattern where one object periodically sends messages to another object specified as its delegate to ask for input or to notify the delegate that an event is occurring. You use it as an alternative to class inheritance for extending the functionality of reusable objects.

Delegate methods are grouped together in either a formal or informal protocol. A protocol is basically a list of methods that the delegate object may choose to implement. If a class conforms to a protocol, it guarantees that the implementation of the protocol methods matches some type of expected behavior. All methods in an informal protocol are optional and a formal protocol may declare optional methods which in turn allow the delegate object to implement only some of the methods in the protocol. The delegate protocol specifies all the messages an object might send to its delegate.

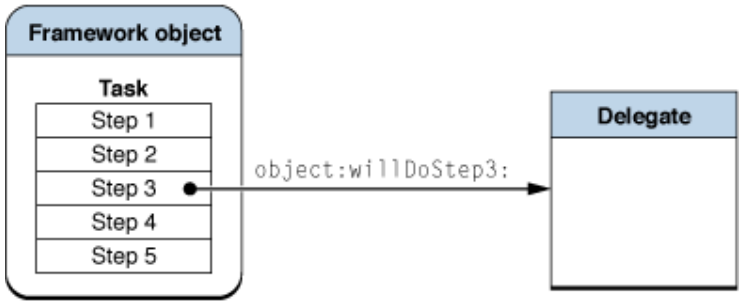

Fig 1: Framework object sending a message to its delegate (Apple Inc., 2009)

3.1.5.2 Model View Controller

The MVC design pattern separates between three distinct types of objects: model objects, view objects, and controller objects. The pattern defines the roles that these types of objects play in the application and their lines of communication.

Model objects represent the application's data and define the logic that is used to manipulate that data. Depending on the application, the model objects will represent almost anything, for example circles and squares in a drawing application.

View objects know how to display data (model objects) and may allow the user to edit the data. The view files are called .xib files and they represent their data with the XML format. The following text is an xml representation of a UIButton with red text named Internal. The XML code is generated by XCode, see the button in the first Pomodoro view.

```
<object class="IBUIButton" id="515009701"><reference key="NSNextResponder"
ref="191373211"/>
      <int key="NSvFlags">292</int>
      <string key="NSFrame">{{20, 319}, {85, 37}}</
        string>
      <reference key="NSSuperview" ref="191373211"/>
      <bool key="IBUIOpaque">NO</
        bool>
      <bool key="IBUIClearsContextBeforeDrawing">NO</bool>
      <int key="IBUIContentHorizontalAlignment">0</int>
      <int key="IBUIContentVerticalAlignment">0</int>
      <object class="NSFont" key="IBUIFont"
        id="546258701">
        <string key="NSName">Helvetica-Bold</string>
        <double key="NSSize">15</double>
        <int key="NSfFlags">16</int>
       </object>
       <int key="IBUIButtonType">1</int>
       <string key="IBUINormalTitle">Internal</string>
       <object class="NSColor" key="IBUIHighlightedTitleColor" id="620590021">
        <int key="NSColorSpace">3</int>
        <bytes key="NSWhite">MQA</bytes>
       </object>
       <object class="NSColor" key="IBUINormalTitleColor" id="617986367">
         <int key="NSColorSpace">1</int>
        <bytes key="NSRGB">MSAwIDAAA</bytes>
      </object>
      <reference key="IBUINormalTitleShadowColor"
        ref="287298069"/>
     </object>
```

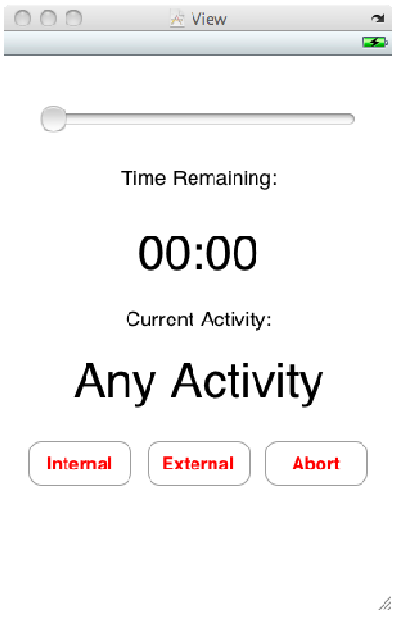

Fig 2: The very first design of our Pomodoro view

Controller objects act as the intermediary between the application's view objects and its model object. Controllers are responsible for linking the models to their views and also to update the state to keep them synchronized. They can also perform set-up, coordinating tasks for an application and manage the life cycles of other objects.

Cocoa uses the Model-View-Controller design pattern as a base to many of their mechanisms and technologies. Therefore, if the user decides to incorporate any Cocoa technology in their own application that follows the MVC-structure, it will not only work better if their application follows the MVC pattern but it will probably be less painless to incorporate the technology.

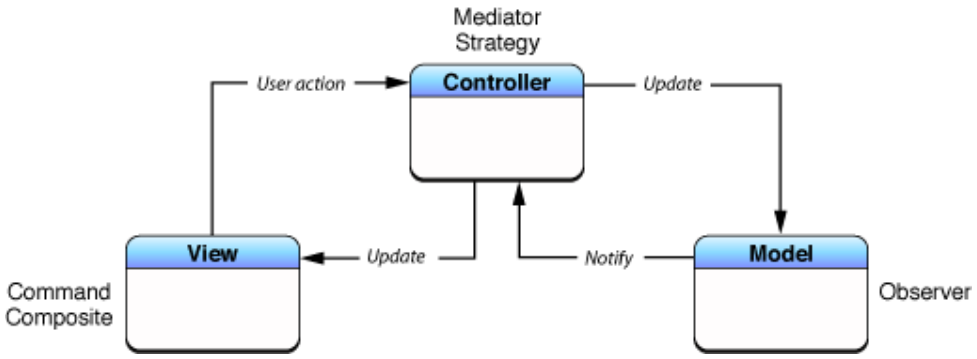

Fig 3: Cocoa version of MVC as compound design pattern (Apple Inc., 2009)

3.1.5.3 Target-Action

The target-action mechanism enables a control object to send a message to another object which in turn can interpret the message and perform it as an application-specific instruction. Typical control objects are buttons, sliders and textfields and the receiving object, the target, is usually a custom controller object.

When the user clicks or activates a control in some way, the information relevant to the event is sent as a message to some target object. In XCode's Interface Builder, it is possible to specify targets for specific events to which controllers will send their messages. It is also possible to set up so that a controller sends messages to multiple targets regarding a specific event.

Given the Pomodoro view displayed in Fig 2. In that application, when the user drags the slider, the slider tells the controller to update its model and view based on the user's input.

3.1.6 Frameworks

In some cases, the frameworks and libraries in the iPhone SDK might not be enough for your specific needs. This section contains a number of frameworks for various purposes that can be used as alternatives to frameworks contained within the iPhone SDK or to write applications in languages other than Objective-C.

3.1.6.1 PhoneGap

PhoneGap is an open source development tool for building mobile apps with JavaScript. It supports development on different plattforms such as Android, Blackberry and iPhone. The purpose is to let the developer develop the mobile apps in HTML and JavaScript while still taking advantage of the core features in the different SDKs of the plattforms. This tool is interesting since it web enables these devices in a consistent plattform which in turn saves the developer from having to implement logic in different software development paradigms. (Nitobi, 2010)

3.1.6.2 iUI

iUI is a framework consisting of a JavaScript library and images for developing iPhone web apps. It is a tool that produces iPhone-targeted web content with the purpose to look and feel like native iPhone apps. (iUI, 2010)

iUI has the following features:

- Create Navigational Menus and iPhone interfaces from standard HTML
- Use or knowledge of JavaScript is not required to create basic iPhone pages
- Ability to handle phone orientation changes
- Provide a more "iPhone-like" experience to Web apps (on or off the iPhone)

3.1.6.3 iWebkit

iWebKit is a file package designed to help the developer create iPhone, iPod Touch and iPad compatible websites or web apps. No prior knowledge of HTML is required and it also includes tutorials to get the developer up and running smoothly. The package can be downloaded free of charge. (iWebkit, 2010)

3.1.6.4 QuickConnect

QuickConnect is a framework that is designed to reduce the developer's application's time-to-market on multiple platforms. The idea behind this framework is that you can write your interface in HTML and CSS and then implement your logic in JavaScript and publish the application on multiple platforms. QuickConnect is an open source framework that is currently being actively developed and there are already a number of applications in the app store that were made using this framework. If

you intend on making an application for multiple mobile platforms, this framework might be a suitable choice. (Barney, 2009)

3.1.6.5 LiquidGear iPhone

LiquidGear for iPhone is a framework that lets you develop iPhone applications using JavaScript and HTML. LiquidGear will act as an intermediary between the iPhone hardware and the web app, letting you access data from the GPS, photo album, calendar and more. (Nitrogen Design, 2009)

3.1.6.6 Appcelerator Titanium

Titanium is a free and open source application development platform which let you create native mobile and desktop applications by writing web code. The logic and UI is written in HTML/CSS/JavaScript when developing for the iPhone, which makes it possible to create native applications without writing any code in Objective-C. Titanium makes cross-platform application development easier since it can compile web code into native application code. (Appcelerator Inc., 2010)

3.1.6.7 Magic Framework

Magic Framework is a native iPhone UI imitator and HTML5 database interface built in HTML/CSS/JavaScript. It's useful for online websites or on-device Webkit-based applications looking to imitate the native iPhone UI and/or use native storage without having to write native code. (Mcfadden, 2010)

3.1.6.8 MonoTouch

MonoTouch is a framework that makes it possible to create C# and .NET applications for the iPhone. Usually, it is not possible to run the .NET run time, but MonoTouch compiles C#/.NET applications into native code that the iPhone will run. (Paul, 2009)

3.1.6.9 Omni Frameworks

Software developers Omni Group have released a number of frameworks for the iPhone. They are mostly extensions to frameworks that already exist and offer more functionality compared to their iPhone SDK counterparts. (The Omni Group, 2010)

3.1.6.10 Cocos2D

Cocos2D is a 2D graphics framework primarily geared towards game development. It has support for many types of transformations such as rotations, scaling and tilting operations among other things. The framework also includes a basic physics engine, touch support and audio support, so it contains most of the necessary pieces required to create games. (Sapus Tongue, 2010)

3.1.6.11 OpenAL

The iPhone supports audio playback through the *Core Audio* framework. An alternative to using the functionality contained within said frame work is to use OpenAL. *OpenAL* is a cross-platform audio API that among other things has support for 3D audio, making it more suitable for games compared to the Core Audio framework.

3.2 Integration

All applications in the Pomodoro Project need to be able to connect to a central "cloud" in order to permanently store a user's information in some sort of database located within the cloud. This section will deal with some ways of accessing databases, authentication and means of sending/receiving messages to service providers.

3.2.1 Communication with Web Service- REST

REST is an architecture style for web services (Wikipedia, 2010). The basic idea is that the web service stores copies of resources that are formatted in the same way and that clients retrieve and store copies of resources within the service. Typically, clients send requests via HTTP with the typical HTTP verbs (although some of the verbs have slightly different meaning in REST and the server sends appropriate responses to the client's request. The resources can be formatted using any type of data description language as long as the same description language is used for every resource and as long as all clients using the service can interpret the data.

Since REST is typically based on HTTP, it is possible to implement as REST client on the iPhone using the *NSMutableURLRequest* class to make HTTP requests (CapTech Ventures Inc., 2009). The data can as mentioned be formatted in many different ways, but two common description languages used in conjunction with REST is *XML* and *JSON. T*he iPhone SDK has functionality for parsing XML with the *NSXMLParser* class and there is also an open source JSON parser/generator called *JSON-Framework* which translates JSON descriptions into Objective-C objects and vice versa.

3.2.2 Communication with Web Service - SOAP

SOAP is a remote procedure call protocol based on XML that is sometimes used in web services (Wikipedia, 2010). A SOAP message is simply an XML file containing information regarding the procedure that is to be called and any possible parameters needed for the procedure. The message can be transported using different application layer transport protocols, although HTTP is recommended since it works well with WWW.

As mentioned earlier, the iPhone SDK contains functionality for generating HTTP messages and generating/parsing XML, so it is possible to create an iPhone SOAP client.

3.2.3 Authentication - OAuth

OAuth is an authorization framework used to give applications access to resources stored within some type of service (OAuth, 2007). This is done by marking resources with tokens where each token represents access rights for one specific application.

OAuth is a protocol based on HTTP messages passed between service providers, end users and applications. The work flow starts with an application requesting a so called *unauthorized request token* from the service provider. When the provider responds with such a token, the end user then has the option to authorize the token. When the token has been authorized, the application can send the authorized token to the service provider and exchanges this token for a so called *access token* which grants the application access to resources stored within the service.

Although the protocol is entirely based on HTTP, there are libraries for various languages that handle the creation and sending of OAuth messages. There are two Objective-C libraries for OAuth that

have been tested to work with the iPhone, *MPOAuth* and *OAuthconsumer.* Both libraries contain functionality that allows iPhone to authenticate towards OAuth services.

3.2.4 Authentication - OpenID

OpenID is authentication solution that provides a way for a user to use a single login to multiple sites and applications (OpenID, 2010). Initially, the user registers an *OpenID Identifier* with some OpenID service provider. When the user then decides to log in to some web service or application using OpenID, the user is shown a form where he is told to enter his OpenID Identifier. When the user has provided his identifier, he is redirected to the service provider where he originally registered his identifier. Here the user authenticates himself by some means, it is up to the service provider to implement the authentication and the necessary strength of it. If the authentication with the service provider was successful, the user is returned to the original site/application and continues with his work.

Currently, there seems to be no existing OpenID library for the iPhone. The protocol is based on HTTP however, so it is possible to wrap OpenID messages in HTTP but that makes things a bit more complicated. Another solution is to have a dedicated site containing an OpenID form which the iPhone application can access through its web viewer.

3.2.5 Network Communication - TCP/UDP

As may be expected, the iPhone SDK also contains classes and functionality for TCP/UDP socket programming. The sockets used by the iPhone have the capability to respond to incoming events such as messages and connections, which makes it possible to avoid polling loops that constantly listen for such events.

3.2.6 Database Access - SQLite

The iPhone SDK contains functionality that allows a programmer to send queries to SQLite databases from code. All SQLite functionality is contained in a library called *sqlite3*. There are also a number of wrappers for SQLite and one noteworthy wrapper is *EntropyDB.* This wrapper is special because it does not require that the programmer has knowledge of the SQL query language and objects can be stored and retrieved through simple function calls, although these functions correspond to rather basic queries.

3.3 Non functional requirements

Requirements not directly related to the behavior and functionality of the application are considered as non-functional requirements. Such requirements may include requirements regarding execution speed, response time and stability. This section will deal with some guidelines on how to improve some performance aspects of iPhone applications.

3.3.1 Conservation of CPU time

One major concern when developing an application is to make sure that the application does not consume CPU time whenever it is idle. This can happen if the application constantly polls for input from the user, which will cause the application to consume all CPU capacity even though the application itself is not doing anything useful. Instead of using polling loops, an event based approach is better. When developing for the iPhone, interfaces are made using the *UIKit* framework

which includes an event based system that reacts to touches and button presses. By following this event based paradigm, you are guaranteed to avoid any polling loops that waste CPU time.

3.3.2 Response Time

The response time of an interface can in some cases be more important to the user than the actual computation speed. Some computation can be hidden in the background if the result of the computation is not immediately needed by the user interface. For example, when a user submits information to a server, the user may not necessarily have to wait for the submission to be completed before he can continue with his work and in this case, performing such an operation in the background might be a good choice.

To improve the response time of interfaces, non critical operations can be made to run concurrently with code controlling the interface or by delaying the computation until it really is needed. In Objective-C, you can create so called *operation queues* which automatically manage the scheduling and execution of operations assigned to the queue. If timing requirements are necessary, there are classical thread constructs in the language that allows you to control the scheduling of operations in more detail.

3.3.3 Stability and Memory Management

In Objective-C on the iPhone, there is no garbage collection or automatic memory management like there is in Java or C#. Errors in memory management can therefore lead to memory leaks and possibly crashes.

In Objective-C the memory space of each object is managed based on a *retain count,* which can be seen as a count on the number of owners to the object. The three main memory management operations in Objective-C is *allocating, retaining* and *releasing.* When an object is allocated, memory is reserved for the object and it's retain count is set to 1. Other functions and processes can apply for ownership of a previously allocated object by performing a retain operation on the object, which will increase the object's retain count by 1. When an owner no longer needs a reference to an object, the owner can perform a release operation which decreases the retain count by 1. A retain count of 0 is an indication that there are no owners in need of the object and in this case, the object is deallocated and the memory space occupied by the object is freed.

Apart from the memory management functionality described above, there is also functionality for *autoreleasing* and *autorelease pools*. An autorelease pool is a collection of objects and when it is released, all the objects in the pool are released as well. Objects are put in an autorelease pool whenever they are marked to be autoreleased by the programmer. When this happens, the object is put in the most recently created autorelease pool and this essentially means that autorelease pools are nested and have a limited scope. For example, if an autorelease pool is created and released within the body of a loop, all objects that are marked to be autoreleased within the loop body are put in the autorelease pool belonging to the loop and released when the loop has finished.

As may be deduced from the above description, problems may arise if the programmer does not keep track on retain/release calls made to objects. It may not be clear either which part of the code is responsible for retaining and releasing the objects. To make this more clear, there is a commonly used convention in Objective-C. If a function returns an object, that object should be set to be autoreleased within the body of the function. The programmer using this function must then make

the decision if he needs to retain the object, should he need a reference to the object outside of the scope of the active autorelease pool. If he decides to retain the object, he is of course responsible for releasing it whenever he does not need the object anymore.

A short example describing how a string object is allocated in the memory and retained by another object as well as released from memory in the end.

```
NSString *example = [[NSString alloc] initWithFormat:@"TEST"];

NSLog(@"example has retaincount: %d", [example retainCount]);

NSString *example2 = example;

[example2 retain];

NSLog(@"example has retaincount: %d", [example retainCount]);

[example release];

[example2 release];

NSLog(@"example has retaincount: %d", [example retainCount]);
```

Gives the following log:

```
example has retaincount: 1

example has retaincount: 2
```

The last NSLog will not print "example has retaincount: 0" since both of the pointers have a total of 0 as a retaincount. This means that the object has been freed from memory and the object doesn't exist anymore.

3.3.4 Performance Analysis and Tools

Along with the iPhone SDK comes a number of profiling applications and other performance analysis applications. This section will only deal with the two most commonly used applications: Instruments and Shark.

Instruments is an application that is installed together with the iPhone SDK which can be used to trace the execution of your application in order to gather information about CPU usage, memory usage among other things in various parts of the application. Instruments also have the capability of recording and replaying sequences of user inputs. This allows you to repeatedly stress test specific parts of an application and analyse specific parts of your code that may cause you problems.

Shark is a profiling application that is also installed together with the SDK. Shark can give you information about the amount of time spent in various parts of your code and also time occupied by the kernel and standard libraries. In certain cases, Shark can give tips for optimizing code should it discover any obvious bottle necks or inefficient stretches of code.

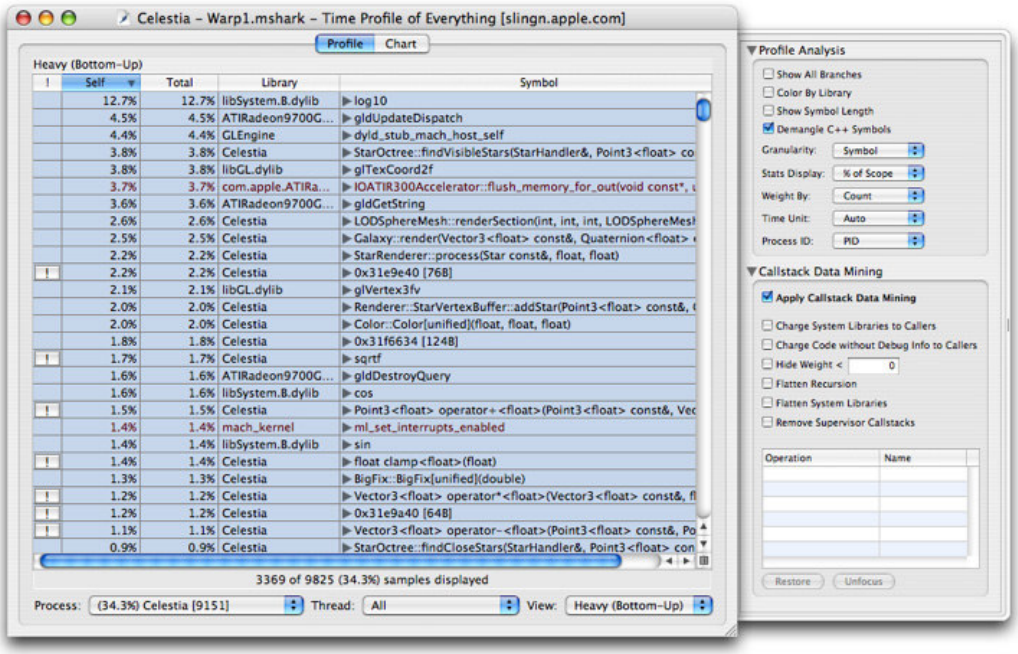

Fig 4: Sample Window with Advanced Settings Drawer visible at right (Apple Inc., 2008)

3.4 Testing

It is widely known that fixing bugs post-development is often much more costly than fixing them during development or using techniques to avoid them altogether. Testing has therefore become an increasingly important part of the development process and this section will deal with what kind of support the iPhone has for testing, either through tools contained within the iPhone SDK or other third-party applications.

3.4.1 iPhone Simulator

The XCode developer environment that comes with the iPhone SDK also has an iPhone Simulator which is the base for all application testing. Testing with this simulator is not entirely conclusive however, since there may be a difference between the performance of an application being run in the simulator and the same application being run on a real iPhone. This is because certain operations run slower on the iPhone compared to Mac OS X and some operations actually run faster on the iPhone.

3.4.2 Debugging

The XCode environment also has a debugging tool. Like many other debugging tools, it has the capabilities of adding break points in your code and step through your code and view values of variables during execution. There is also a debugging console to which an application can write messages and crash logs are also stored.

3.4.3 Unit Testing

XCode has support for unit testing. Unit tests are written as subclasses to a generic class for testing called *SenTestCase* contained within the iPhone SDK. This interface contains functions that can be used to report errors to the XCode environment. When the unit tests are run, the results show up in the debugger console.

3.4.4 Automated Builds

Continuously building a project is a good way to ensure that any changes made to it integrate well with the rest of the project. Automated build tools can also report what specific changes broke the project.

Hudson is a well known tool for automated builds that works together with some common version management systems such as SVN and CVS. Hudson is capable of creating so called "freestyle builds" where the user can tell Hudson to execute a build script whenever an automatic build is made. By creating a script that runs XCode's own build executor, *xcodebuild*, on the code that you want to automatically build. Since it is necessary to build iPhone applications on a Mac, you may have to build the application on a computer other than the one running Hudson. This is not a problem however, since Hudson supports remote builds and you can remotely build applications on other computers, including Macs. (Nachbaur, 2009)

3.5 Maintainability

The MVC design pattern was created to enable portability and maintainability. This design pattern supports maintainability because it is a program structure which separates the *model*, all the data, from the presentation layer, the *view*. Future maintainability of the system becomes rather harmless since, changes in the view doesn't affect the management of data, and data may be reorganized without worrying about changing anything in the View as long as the interfaces that the View uses for accessing data remain the same. To connect the model together with the display and user interaction, MVC introduces the intermediary *controller*.

Objective-C was designed to have a high maintainability of large code bases. One way to improve code is to break it down into smaller pieces. To cope with this problem, Objective-C borrowed and extended the concept of *categories* from the language SmallTalk and a category collects method implementations into separate files. This is usually used to group together methods with similar purpose or functionality, for example arithmetic operations such as addition, subtraction, multiplication etc. could belong to one universal arithmetic category. Categories can be seen as an alternative to sub classing since Objective-C categories provide means to add methods to a class. If you make any class a part of a certain category, all methods belonging to that category will be added to that class and all its subclasses. This means that you can extend a class without having access to its source code since category methods are added to a class at runtime.

One maintainability issue that exists when developing on the iPhone is the fact that it is a closed platform. There is no possibility to change anything in the iPhone SDK, or at least it is not allowed or recommended in any way. However, there is a technological way around using the public API:s and use non-public methods instead. Apple is unfortunately very aware of what a developer is expected to be able to do with their API and they won't allow any application which uses any non-public method in their App store. Developers are allowed to incorporate other libraries for their own purposes but creating an own include file which links code to the non-public parts in the API or against the hardware is forbidden.

3.6 Security

As the world becomes more and more connected to the Internet, computer security threats become an increasingly important issue. Important aspects of security include hacking/hijacking, data

integrity and authorization issues among other things. This section will deal with some of these security issues and how they are resolved on the iPhone.

3.6.1 Restrictions on Code Execution

One common way for hackers to hijack an application is to cause a buffer overflow. This can happen when an application does not check the size on the input data to see whether the input data is as large as it's allocated space of memory. If this is the case, a hacker can give an application input data that is larger than it's allocated space, which allows the hacker overwrite memory space with executable code. If the overwritten memory space belonged to another application, the hacker's code will be run at some point when the application is supposed to resume execution.

On the iPhone, the above exploit is weakened by using a "sand-boxing" mechanism. The basic principle is that a sand-boxed application is only able to write data to files that belong to the application. By doing this, it is impossible for an application to overwrite files belonging to the system or other applications. Sand-boxing could also involve limiting other system resources, such as denying an application the ability to access any networking features. Whenever an application is run on the iPhone, it is sand-boxed automatically. It is important to remember that a sand-boxing mechanism does not prevent buffer overflow attacks; it only limits the damage that can be done by such an attack.

The only way to truly prevent buffer overflows is to follow some coding guidelines that makes your code safer. This includes using safer string operations that take string lengths into account and conventions for checking buffer sizes among other things. For more details, check the *Secure Coding Guide* in the iPhone OS Reference Library.

3.6.2 Secure Network Communication

The *CFNetwork API* in the iPhone SDK contains functionality for using special types of sockets which have extended functionality compared to basic sockets. This includes the ability to use the SSL and TSL security protocols for encryption, authentication and data integrity.

3.6.3 Key Chain

A typical user usually has many accounts to various services. Having the same password to every account is not a good idea, but remembering many different passwords can be difficult for a user unless he documents his passwords in some way. Both approaches are serious security hazards.

Mac OS X and iPhone OS has a solution to this problem, known as the *key chain*. The key chain is an encrypted container that stores login information for various services. The keychain has two states, locked an unlocked. In the locked state, the key chain is encrypted and no application can access the information stored and in the unlocked state, trusted applications can read the information in the key chain. On the iPhone, the key chain is automatically unlocked by applications, and the application is only capable of accessing its own information. This makes logging in to various services transparent to the user. The iPhone SDK has libraries that allow the programmer to retrieve and store information in the key chain.

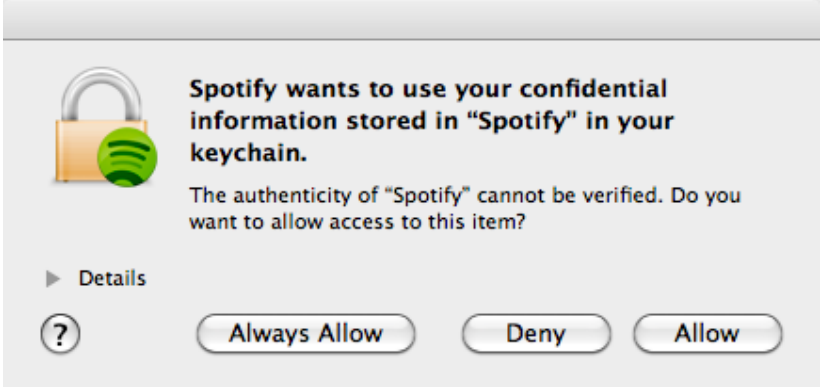

Fig 5: Key chain example

3.6.4 Viruses and Malware

To this day, there exists no antivirus/malware tool for iPhone (Zorz, 2009). The reason for this seems to be that there is still no way for an application developed with the public iPhone SDK to access resources outside of its own sand box("sand-boxing" described earlier), which is necessary to detect a virus outside of an applications scope. On the other hand, the few iPhone viruses/malware that are circulating have so far only been able to target jail-broken iPhones. The simple reason for this is that by jail-breaking an iPhone, the user is capable of installing third party software not approved by Apple. This of course means that a malicious attacker is capable of installing malware as well.

In essence, viruses and other malware are currently not an issue for non jail-broken iPhones, but this may change in the future.

3.7 Interface and User Experience

Apple has for a long time been an innovator within interaction design and is known for highlighting the importance of a good user experience. To ensure a good user experience, Apple has created a set of guidelines with the purpose of creating consistency and ease of usage across all applications. The following section will illuminate the reader with input methods on the iPhone both physical and non-physical, design patterns that are commonly recurring in the menus and in the applications and common interface components used in Applications on the iPhone. Finally, there is a small comparison that scratches the surface between iPhone and Android platforms.

3.7.1 Input - iPhone

There are four buttons on the iPhone: power on/off, volume up/down, silent mode switch and the *Home Button.* The three first buttons are self explanatory, the fourth button is iPhone/iPod specific. It is mainly used as a shortcut that allows the user to return to the *Home Screen* at any given time. The home screen is the main hub of the iPhone interface and it contains shortcuts to applications, settings and other utilities. Whenever a user is running an application and presses the home button, the application immediately exits and returns him to the home screen. The state of the application is not stored in any shape or form, meaning that any work or progress done within an application is lost if the user presses the home button unless the application specifically saves information in some form. The home button also reacts to double presses and a double click will either bring up a list of recently phoned contacts or a search field depending on the user's preferences.

The most prominent input mechanism on the iPhone is the touch screen. The iPhone can tell the difference between clicking and dragging, giving it capabilities similar to a regular computer mouse. Since there are no hardware buttons used for controlling the iPhone interface outside of the home button, nearly all of the interaction with the iPhone is done via on-screen buttons and widgets controlled by the touch-screen. One common widget is a virtual keyboard that is used for all text input to the iPhone interface.

Finally, the iPhone has a 3-axis accelerometer which can be used to sense how much the iPhone is tilted. It is not as commonly used as the touch screen, but one common usage of the accelerometer is to switch between portrait and landscape mode. The accelerometer is however commonly used in games for various purposes.

Fig 6: iPhone and HTC Magic (One of many Android models) (Apple Inc., 2010), (Vodafone, 2010).

3.7.2 Interface Patterns

According to Tidwell (2005) patterns are structural and behavioral features that improve the "habitability" of something - a user interface, a web site, an object-oriented program, or even a building. Patterns are useful to make things easier or more accessible, they can be seen as a description of best practices within a given design domain.

3.7.2.1 Hub and Spoke

The iPhone's home screen follows an interface pattern called Hub and Spoke.

The Hub and Spoke pattern isolates the sections of the app into mini-applications, each with one way in from the main page.

Fig 7: Home Screen iPhone (Apple Inc., 2010)

According to Tidwell (2005), the Hub and Spoke pattern is suitable if your UI contains several discrete tasks, sub-applications, or content elements such as forms, demos, games, articles, transactions or other self-contained tools. It is also suitable when you want to reach all of them from one central page or screen. However, Tidwell (2005) argues that you don't want to connect each of them to all of the others and some of the possible reasons she mentions are:

- To reinforce the separateness of these sub-applications
- To restrict workflow to force the completion (or explicit cancellation) of a task
- To eliminate visual and cognitive clutter
- Physical space constraints

Furthermore, this pattern is suitable for smaller devices because it tells the user to focus on one section at the time and if the user desires to move another section, he must move back to the hub and pick a new spoke. Also, the restricted navigation paths might prevent errors since users have less opportunity to get into a confusing state. Take the iPhone as an example, there is almost always three available options, drill down, go back one window or head directly back to the hub. Even though some users may find this kind of navigation a bit annoying, especially if they are used to free form navigation, it still carries advantages. The fact that it is very clear for users how to orient themselves to the hub and different spokes is really powerful.

Moreover, this kind of navigation pattern means tighter control over what the interface needs to accomplish. In turn, it might result in a cheaper or simpler implementation since it is easy to add more spoke links to the hub page whenever extra functionality is added to the application.

3.7.2.2 One-Window Drilldown

Another interface pattern commonly used on mobile devices is the O*ne-Window Drilldown* pattern. It is commonly used to display data or views with hierarchical ordering. When using this pattern, the

display is only used to display one level of the hierarchy at a time and whenever the user decides to traverse up or down the hierarchy, the next level will occupy the entire screen. This pattern is suitable on mobile devices because it is often not possible to display many panels or windows at a time due to size limitations. It is also a simple pattern which focuses the user's attention in one place and all the options available at each stage in the interface are very clear.

The one-window drilldown pattern is commonly used on the iPhone. Menus are often displayed as lists of items and whenever a user selects an item in the list, the current view is replaced with a new view for the selected menu item.

Fig 8: The settings menu on the iPhone. > indicates that selecting this item will open up a new window with new options (Apple Inc., 2010).

3.7.3 Common Interface Components

Much like in the Java Swing API, the UIKit library contains a number of standardized interface components commonly used in iPhone applications. The advantage of using such standardized components is familiarity. Since the standard components are commonly used in other iPhone applications, users intuitively know how these components are used as long as they are implemented in such a way that they match common behavior patterns.

The most basic interface widget on the iPhone is the *status bar* which is always located at the top of the display. This bar displays the network connectivity, battery status and the current time. The developer has the option to hide the status bar when executing his application although this is usually not such a good idea unless the status bar distracts from the visual experience, for example when playing a game.

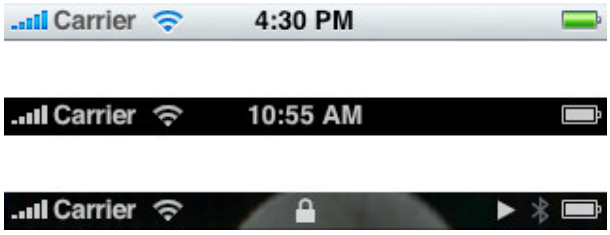

Fig 9: Status bar types (Apple Inc., 2010)

Another common interface widget is the *navigation bar,* located right below the status bar. Typically, the navigation bar displays a label telling the user where he is currently located within the interface hierarchy, an option to return to where the user previously was and possibly some buttons that enable the user to interact with the objects in the current view. Common interactions include the ability to add a new item in a list, editing an item, deleting an item and saving current settings among other things. There are many standardised button icons used for these purposes. For example, a '+' icon signals to the user that the button is used for adding an item to a list. Using such standardised icons further increases the familiarity of the interface.

Fig 10: Navigation bar example (Apple Inc., 2010)

If the user should be able to perform many actions within the current context, it may be suitable to add a *tool bar.* A tool bar is a horizontal bar containing icons representing various actions located at the bottom of the screen. Just like the navigation bar, there exist standard icons for common actions that should be used to the largest extent possible.

Fig 11: Tool bar example (Apple Inc., 2010)

If your application supports many modes of operation or many different views, a *tab bar* can be used to quickly switch between views. If the tab bar is visible, it will be located below the tool bar and there are standardised icons for tab bars as well.

Fig 12: Tab bar example (Apple Inc., 2010)

As mentioned earlier, views and menus are commonly ordered hierarchically in lists. On the iPhone, such lists are called *table views* and they can be used to display lists of objects or a list of menu options. If selecting an item in the list opens up a new view, this is usually signaled with a '>' sign in the item row. These types of lists can be used to enforce a one window drilldown as described previously.

3.7.4 iPhone Vs. Android

Android is designed to support multiple interactions modes. For instance, it can be run on a large screen touch-only device or a small screen device with 4-way navigation keys or both. A great deal of effort has also gone into making G1 Android UI as consistent as possible and as an Android developer you are given a lot of freedom creating new applications and services. However, maintaining a high level of consistency in the interface and user experience is hard due to the fact that Android is an open source platform. Open source means that it is completely up to the developer how the application should look and behave, and it should be since it is open source. The risk is that many of the new applications will follow a new pattern which might confuse the user. Apple on the other hand encourages a tight relationship between the design of the individual application and the look and feel of the underlying platform. This has its advantages in a way that the interface gets consistent and predictable which in turn leads to a user which knows how to operate it. However, you can still customize your own interface with the iPhone SDK but there is more work to it. A good key rule to have in mind before customizing your own interface is: if you can't design something that is better than Apple, then you should probably reconsider your options.

One big difference between the iPhone and Android is that in general, designing an application for Android means designing without knowing all the specifics, such as what the configuration device will be or even knowing which device. It is therefore necessary that an Android application supports user behavior, assumptions and interactions for both touch and non-touch devices whereas designing for the iPhone the designer can focus on a single device. This is of course one reason as to why interface designs tend to be more inconsistent for Android.

Furthermore, Apple provides rather extensive information about how to organize your interface, such as when to use common interface components and when they should be used. For instance, in iPhone Human Interface Guidelines Apple describes why you should use one navigation control instead of a multi-segment back button in the navigation bar followed by arguments why this is the case.

Fix 13: Navigation bar containing a single navigation control (Back to *General*) (Apple Inc., 2010)

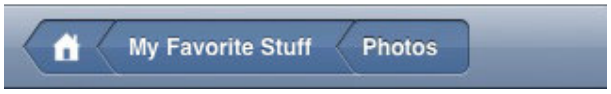

Fig 14: A multi-segment back button, not recommended (Apple Inc., 2010)

Google provides guidelines over different controls and components and how to operate them but they lack information about how they should be used in the interface.

4. Implementation & Execution

Apart from the pre-study and evaluation of the iPhone platform, one large part of the Pomodoro project is to implement a reference application, namely the Pomodoro Application. We therefore had to make decisions on how to go about designing the application, both in terms of code and interface design, and this chapter will deal with these issues.

4.1 Application Design

This section will describe the underlying design of iPhone applications and some design details specific to the Pomodoro Application.

4.1.1 Views, View Controllers and Navigation Controllers

The three main elements in an iPhone application are *views*, *view controllers* and *navigation controllers*. Views define the appearance of various parts of an application and also interactive view elements such as buttons, sliders and table views among other things. Views are typically not defined in code, they are often designed using Apple's *Interface Builder* which is a graphical tool used to create interfaces.

View controllers on the other hand define the underlying behavior of the application when interacting with the view elements and act as delegates for views. For example, the view controller can define exactly what happens when a user presses a specific button. In practice, an interface event (such as a button press) is assigned a delegate method in the corresponding view controller which is called whenever the event arises. This can be done in Interface Builder or in code depending on what is most suitable for the situation.

Finally, navigation controllers handle the navigation between various views in an application. A navigation controller can be seen as a stack of views and view controllers and the navigation controller is responsible for handling the pushing/popping of views on the stack. The advantage of using such a stack is that it is very simple to traverse backwards in the view hierarchy since all you have to do is pop the active view and it is just as easy to push a new view on the stack in order to traverse forward to another view.

The underlying code structure of the Pomodoro application mirrors the structure described above. Each part of the application has its' own view designed using Interface Builder, a corresponding view controller class defining the behavior of various interactions made with the view and a navigation controller handles the traversal between views. As will be described later, the Pomodoro Application is based around a number of tabs and each of the tabs has its' own navigation controller with its' own stack of views.

4.1.2 Singletons

Various parts of the Pomodoro Application need to access the same information, for example, a large part of the functionality of the application revolves around the three main activity lists used in the Pomodoro technique. The solution that is used in the Pomodoro Application is to use singleton classes for data and/or functionality that need to be used by many parts of the application. For example, the three activity lists are represented by a singleton which in turn contains the three lists and all network communication is also handled by calling various methods on a singleton object. This is convenient since there is no need to pass around pointers to the same information and there is no need to create the same objects over and over again to reuse functionality.

4.1.3 Model

The domain model used in the application is based on an older domain model (see Appendix 5) and is largely focused around the three activity lists used in the Pomodoro Technique: 'to do', 'activity inventory' and 'unplanned and urgent'. As mentioned previously, these three lists are represented by a singleton class. Each of the three lists contains a number of activities and each activity contains a number of references to Pomodoros that have been completed while performing the activity. All Pomodoros contain references to interruptions that have occurred during the course of the Pomodoro and these can be either internal or external interruptions.

Finally, there is also a singleton for the active Pomodoro which the user is currently running. The reason for making this Pomodoro is simply because there should not exist more than one active Pomodoro at any given time.

4.1.4 Design for Multiple Language Support

A non-functional requirement for the Pomodoro Application is to be able to support at least two languages. To solve this, we have stored text elements in so called property list files, which is an Apple proprietary format that has similarities with XML. The property list contains key-value pairs for every text element in the user interface and by creating different property list files for each language with the same keys but different values, you can switch between languages by simply switching the property list from which the application loads its' text elements. This means that you will have to translate the original property list file to support new languages, but the advantage is that you actually don't have to rewrite a lot of code in order to support more languages.

4.1.5 Networking and Handling Offline Usage

As previously mentioned, the Pomodoro Application must communicate with a central web service known as the cloud. The cloud is a REST web service which uses OAuth for authorization and authentication.

The general idea behind the networking solution that was originally intended to be used in the Pomodoro Application is that the application loads all of the data upon login, make all changes locally during execution and finally commits all changes to the cloud upon shut down. The purpose of this design is to increase the response time of the interface during execution. Since all data is loaded upon login, the application never has to query the server for data during execution and there are no waiting times associated with retrieving data from the cloud.

There are a few scenarios that cause problems with the above solution. One of them is if the user manages to log in and download all data during startup, but somehow loses his internet connection before shutdown. Using an unmodified version of the solution above, it would mean that all modifications that the user has done between login and shutdown will be lost. The solution that is used to solve this problem is to store objects that need to be modified, added or deleted on the server in a local database if the user for some reason doesn't have an internet connection when he exits the application. When the user logs in with an internet connection the next time, the server will commit all the necessary changes to the server. There is a chance that there will be synchronization errors if the user does modifications with another client before he logs in and commits all the changes to the server, but we warn the user of this if he doesn't have an internet connection upon shut-down and hopefully the user will synchronize his information before using another client.

Another scenario that causes problems is when the user doesn't have an internet connection during the entire execution of the application. Since authentication is entirely online-based, there is no way of verifying who the user is. If multiple users use the same iPhone, there would be no way of telling whose information should be loaded when the application starts either. The solution that is used in the Pomodoro Application is to have a public offline account whose data is stored locally in a database. This will allow the user to use all of the functionality of the application even when the user is offline. The obvious disadvantage is that the user cannot continue work that has previously been done online, but at least there is some way of using all functionality offline.

We started developing the above solution but realized that it was rather complicated and we felt that we wouldn't have time to test it extensively. What we settled for in the end is a very simple solution with an offline mode and an online mode. When using the online mode, the client downloads all data associated with a user upon login and then performs all changes done during execution as synchronous requests to the server. When using the offline mode, the user uses a special local account which interacts with a database that stores all data locally. This means that a user can still use the application and the Pomodoro technique without an internet connection, although there is no way of altering data stored in the cloud offline.

4.2 Interface Design

A smart and user-friendly design is important for developing a successful product. Designing the user interface for the Pomodoro Application has been a large part of the iPhone Pomodoro project and the following piece describes how the design process was carried out and the result of that process.

4.2.1 Methodology

In the early stages of development, we created interface sketches with pen and paper after a thorough investigation of the different interfaces found among the standard applications in the iPhone and guidelines according the iPhone Human Interface Guidelines document. Apple provides a couple of standard ways to design your application and the most common way is to use the one-window drilldown pattern or a tab bar for navigation. As mentioned earlier, there are of course ways to customize your own interface but remember, this will require much more work and you should probably be a skilled designer to be able to make something better than Apple. By following Apple's interface guidelines, you also improve consistency with other iPhone applications which will make the interface more intuitive for users with previous experience with the iPhone.

The first paper design was a paper sketch over the Pomodoro Application made completely with the one-window drilldown pattern. Some screens such as the Pomodoro timer were not fully developed and were only included in the design to show its purpose in the application. The only things that we had planned to change in the future with these views were the aesthetics, all the functionality that was conveyed in the original sketches would remain the same. The second paper design had a tab bar in the bottom of the screen which would act as the main navigation mode of the application. Each tab represented a main view and the one-window drilldown pattern was used under each tab to cover all functionality of each view. The final and third paper design combined the one-window drilldown paper sketch with a toolbar placed in the bottom of the screen. The difference between a toolbar and a tab bar is that the toolbar contains buttons that act on elements in the current view rather than the tab bar which is used to switch between different modes or views in the application.

The toolbar would naturally change buttons according to which view that is currently selected and it would not necessarily be present throughout the whole application.

Based on the sketches we made, we used a method called Paper Prototyping, which is a useful design method to perform usability testing in the early stages of development. It is useful to get early feedback on the design without implementing a single line of code or constructing anything other than the paper prototype. This method can be applied on virtually any type of interface such as desktop applications, hand-held devices, web sites and many other types of hardware devices.

Furthermore, paper prototyping is conducted on test subjects (common users) by giving them a realistic task to perform by interacting with the paper version of the interface. The paper prototype is controlled by one person playing the computer, who doesn't explain how the interface is intended to work. By taking notes of how the users behave while performing the given task, flaws and errors in the design will present themselves making it possible to remove them at an early stage of the development process. The results of the tests carried out with the paper prototypes are described more thoroughly in section 4.3. However, the application was designed using our tab bar design in the end.

4.2.2 Navigation

Navigating within the Pomodoro Application is done by using a tab bar which is situated at the bottom of the screen. It has the total of five tabs, which is considered the maximum number of tabs that can be displayed all at once in the tab bar. If there are more than five tabs, the fifth tab will constitute a 'more screen'. This tab takes the user to a new screen where the users can select their favorite tabs to be displayed in the tab bar. Since the application has five main options, there was no need to use more than five tabs, one for each main option (Activities, Pomodoro, Plan, Statistics, Settings).

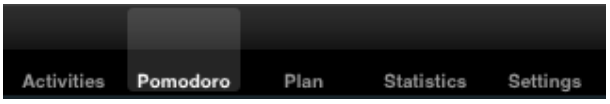

Fig 15: Tab bar with the main options

The tab bar should be accessible from anywhere within the application to not confuse the user. However, the Pomodoro Application makes one exception to this rule. During an ongoing Pomodoro, the user has the option to create either an internal or external interruption and at the same time add a new activity. If this option is selected by the user, a new view will cover the current Pomodoro view, including the tab bar. The reason for hiding the tab bar in this situation is because the user has to complete this task before continuing using the application.

To complete the tab bar, it is necessary to add complementary icons in each tab to make it even more user-friendly and intuitive. However, the icons should be designed with graphic design in mind, that is, they should provide a visual representation to express the meaning each tab more clearly to its users. These five icons have been chosen to convey the meaning of the functionality contained in each tab.

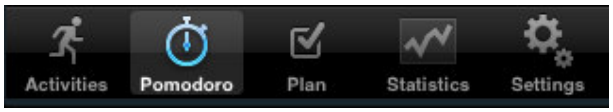

Fig 16: Tab bar with icons

Icons for each tab:

- Running man, shows a person performing an activity.
- Stopwatch, represents the Pomodoro timer.
- To do, describes the plan the day feature.
- Line chart, shows the statistics option.
- Settings, the cog wheels are similar to Apple's own standard icon used for representing settings.

4.2.3 One-Window Drilldown

Under each tab, the user will be able to drill down to reach his desired destination. In the Pomodoro Application, there are at most three levels in any given drill-down. The following images shows how the user uses the one-window drilldown pattern to drill down starting from the 'Settings' tab in the order 'Pomodoro Settings' > 'Pomodoro Length (minutes)'. The final end screen is the edit view for editing the length of the Pomodoro timer.

Fig 17: First window under the Settings tab. Pushing 'Pomodoro Settings' drills the user down one window. Pushing 'Pomodoro Length (minutes)' drills the user down a second time, reaching the end screen

4.3 Testing

Testing is an important phase during all software development. The testing phase must ensure that the product functions correctly and that the usability of the product is at an acceptable level. The

following section will show how the interface testing was conducted as well as how the Pomodoro Application itself was tested.

4.3.1 Interface Testing

Interface testing was done using the paper sketches that we had developed during our design phase. Based on these sketches, we developed paper prototypes of these interfaces by writing sticky-notes that correspond to parts of the application state and these prototypes were used in user tests involving seven people. For each interface, the user was asked to perform a number of tasks and the tasks were identical for each interface. An example of such a task would be to add an activity to the Inventory list. The only information that was provided to the user before the testing began was an introduction to the domain and an introduction to the Pomodoro technique. During testing, one of us played the computer and the other noted down unexpected user behavior.

The general consensus among the users was that the tab design was better compared to the pure one-window drilldown designs. The users felt that being able to access all parts of the application without returning to a central hub was preferable due to faster access. Some users thought that being able to use common functionality wherever you are (for example being able to add an activity using a '+' button in any view) in the toolbar design was nice. Overall, users appreciated the possibilities of manipulating activities using similar buttons located in the same place in the toolbar. Despite the positive feedback that the users gave on the toolbar design, they still thought that the one-window drilldown navigation of the toolbar design was too restrictive and that it was preferable to have tabs for navigation instead of a toolbar for quick manipulation. However, due to this feedback, we decided to alter the tab design so that the user is able to create activities in suitable situations to mimic some of the functionality that is provided through the tool bar.

One common flaw in all of the designs was that the users could not see the purpose of prioritizing activities in the Plan the Day view. We suspect this is because the advantage of prioritizing your To Do list is not immediately visible in the view itself, it will only be apparent when the user selects an activity for a Pomodoro and notice that the highest prioritized, non-finished activity is pre-selected. We have not found a good way to convey this in the Plan the Day view itself, but we hope that the user will learn through experience that the highest prioritized activity is pre-selected in the Select Pomodoro view.

4.3.2 Pomodoro Application Testing

The application has been continuously tested throughout development. Whenever a larger amount of functionality had been implemented, it was tested thoroughly to make sure that it integrates well with the rest of the application. Testing has mainly been done with the iPhone simulator, but the application has also been tested on an iPod Touch. From our experience, you can expect that if the functionality works in the simulator, the same functionality will also work on a real device. However, performance will most likely be different, but also the appearance of some interface components can also differ between the device and the simulator.

5. Result

This chapter will present a walkthrough of the interface and motivate some choices regarding the interface design.

5.1 Interface Design

This section will present a walkthrough of the interface and explain why certain choices of the interface design was implemented in the following manner.

5.1.1 Interface Walkthrough

Common solutions that are recurring for the whole interface are:

- In table views, '>' indicates that the option is pushable and that it will drill down the user further into the interface. This means that the whole screen will be replaced with a new screen in the next level of the hierarchy. If the new view located at the bottom of the hierarchy, the view will always contain a 'Cancel' button at the left and a 'Done' button at the right. Clicking any of these two buttons will take the user back to the previous screen. Otherwise the navigation bar will most likely only contain a 'Back to previous view' button.
- Any table view cell with '>' together with a value indicates an option that will take the user to an edit view. This view will use a keyboard or number pad to give the user the ability to give the right input depending on the view.
- Any operation that could potentially be destructive in any way is always carried out with a safety question before the operation is executed.

Start Screen

The start screen is the first screen and the entry point to the Pomodoro Application. The user is presented with two options, one button that says 'Log In' and another that says 'Work Offline'. The 'Log In' button takes the user to a new screen where he is asked to provide credentials before he can connect to the service provider and continue using the Pomodoro Application with a private account. Furthermore, 'Work Offline' is the easiest way in to the Pomodoro Application however, work offline is only private to some extent given that the phone is private.

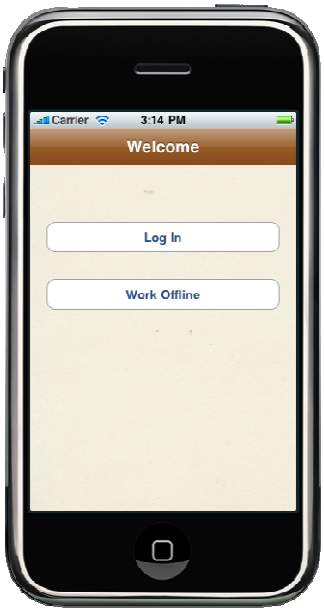

Fig 18: Start screen

Finally, the start screen is supposed to have clear entry point/points and clearly state which options the user has before entering. This is something the Pomodoro Application provides in a simple manner.

Activities

Once the user has passed the start screen, he encounters the activities view. The activities tab is always the pre-selected tab once the start screen has been cleared. At the top of the screen in the navigation bar, the user has the options to choose which activity list he wants to populate the screen ('To Do', 'Inventory' and 'Unplanned'). The user is also given the option to add a new activity by pushing the add button (+). Depending on which tab in the navigation bar that is currently selected, the new activity will be added to that list. Also in the activities view, the user is able to click on any activity or the 'Delete Completed Activities' button, which removes any activity that is marked as completed.

Adding a new activity is started by pushing the '+' button. Once the button has been pushed, the user is presented with a new view called 'New Activity'.

Fig 19: Step by step adding an activity

Furthermore, if the user clicks on an activity, he will traverse one step down in the activities hierarchy entering the detail view for a single activity. This view is a grouped table view, one group for the information about the activity itself and another group for buttons that can be used for moving or deleting the activity. This view was implemented in this way to separate the detail information about the activity from the moving/deleting options.

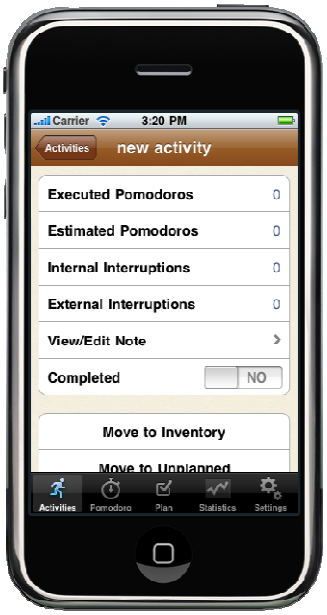

Fig 20: Detail view of an activity

Pomodoro

Before the user can use the Pomodoro view, the user must choose an activity because it isn't possible to start a Pomodoro without it being connected to an activity. Therefore, as a safety measure, the first view being presented for the user under the 'Pomodoro' tab is the 'Select Activity' view. However, if a user has chosen an activity already, this view will be skipped.

Fig 21: Select activity screen

The Pomodoro view will most likely be the view that the user will spend most of his time while using the Pomodoro Application. This screen has a large segment displaying the Pomodoro timer. The

timer has the color red when a Pomodoro is ticking and green during breaks to point out the difference for the user. Right below the timer segment, there is a label which displays the current name of the activity and below the label there is a smaller segment with two buttons 'Start' and 'Abort'.

Fig 22: Pomodoro timer for activity and break

Furthermore, once the user pushes the 'Start' button, the segment updates itself to become three button segments named 'Internal', 'External' and 'Abort'. The 'Internal' and 'External' buttons are used to report internal and external interruptions. There are two different options when reporting an interruption independent of which kind of interruption it is, namely 'Report' and 'Report & Add Activity'. The 'Report' button increases the interruption count while the 'Report & Add Activity' button increases the interruption count and pushes and presents a view that allows the user to create a new activity. This view is very similar to the 'Activities' view, except that the user can't add a new activity under the to do list. This constraint is there because the application follows the rules of the Pomodoro technique. Since there is no need to report interruptions during a break, the segment will only display 'Abort' during break time.

Fig 23: Shows how to add interruptions and how the segment control changes

There are two labels above the tab bar, the first one is called 'Internal & External' and it displays the counts for internal and external interruptions connected to the current activity. The second one is called 'Executed Pomodoros' and it displays the number of Pomodoros the user has executed for the current activity out of how many Pomodoros the user estimated that the activity would take to complete.

Finally, when the Pomodoro timer reaches zero, a pop-up will present itself in the center of the screen asking the user if the activity was completed or not.

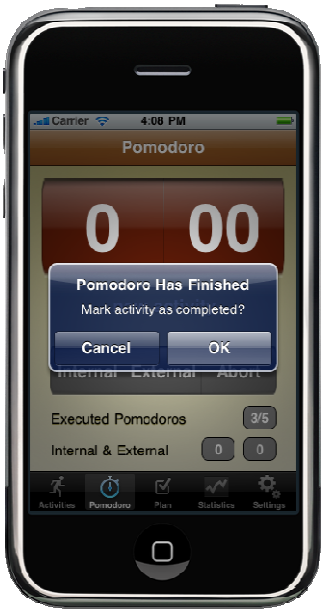

Fig 24: Pomodoro has finished

Plan The Day

'Plan the Day' is a grouped table view with three groups. In the first group, the user can select how many Pomodoros he intends to complete for the whole day and select activities that are to be completed during the day. Notice how they both have the '>' sign, which means that there will be a sub-view where the user will complete these options. If the user selects the 'Select Activities' option, the user will be given the ability to select activities from all three lists that will later constitute his To Do list. The second group is the 'Current To Do' and This group displays the activities in the To Do list. The last and third group is a label that shows the user how many Pomodoros he has executed out of the number of Pomodoros he planned for the whole day.

Fig 25: Plan the day

Finally, the navigation bar holds one different option in the 'Plan the Day' view, the 'Prioritise' button. When the user pushes the 'Prioritise' button, the view goes in to a different mode which allows the user to drag and drop the activities which are listed under 'Current To Do', making it possible to prioritise them in any desired order. This is useful because when the user selects an activity for a new Pomodoro in the 'Select Activity' view mentioned above, the marker will be pre-set on the activity which is topmost of the list and will work its way down as the activities becomes completed. The second button in the navigation bar is the add new activity button (+).

Fig 26: Prioritise activities in the current To Do

Settings

The 'Settings' view is very basic and consists of a one-window drilldown which lets the user alter various settings.

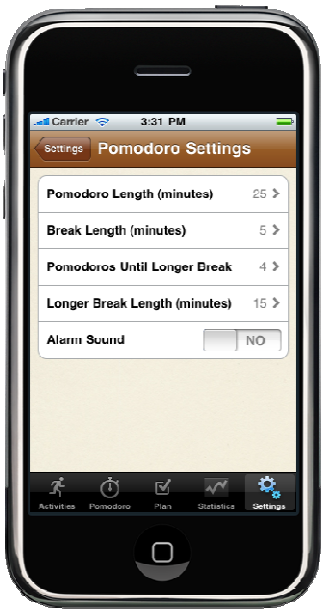

Fig 27: Settings

6. Conclusion & Lessons Learned

The Pomodoro project had an extensive research phase which prepared us for development with the iPhone SDK as well as the techniques and interface designs patterns associated with the iPhone. However, developing for the iPhone during this project has not been entirely flawless. First and foremost, the process of getting Interface Builder to work well together with our code was hard to understand at first. After having worked with Interface Builder in this project, we have come to the conclusion that it is only suitable for simpler views. If your view needs to completely alter its appearance during execution, you will not be able to do it with Interface Builder. Also, the naming conventions for functions and such in the iPhone SDK are in our opinion not that good. There were far too many occasions where we had to look in the Apple documentation to find the methods and properties we were looking for. We have also realised that the simulator is not sufficient to test with. The simulator does not only give an incorrect estimation of performance, there are also instances where the appearance of graphical components can differ between the simulator and the actual device. In general, the tools that are used to develop iPhone applications are in our opinion not as good as for example IDE's and tools developed by Microsoft, although this does not detract much from the experience of developing for the iPhone.

In this project, just like many others, it is important to test the interface as much as possible before implementing anything since it is easier to change the interface in an early stage of development before you get too committed with the decisions you make. To ensure that the product has high usability and user-friendliness, there exist a number of design methods that can help anyone already in the earliest stages of the project with creating interfaces as well as conducting user tests. These methods are useful tools to apply in various kinds of projects, not just software development projects. By using these kinds of methods, you are less likely to have to change your interface design at later stages of development.

By doing this project, we have concluded that you should try to use the most common interface patterns supported by the iPhone SDK as much as possible. The reason for this is simply ease of implementation. If you choose to use a one-window drilldown or a global navigation design with tabs, there is already a lot of support for this and it is easy to implement such an interface quickly. On the other hand, if you wish to create an interface of your own that looks nothing like the interface components supplied by Apple, you are forced to write everything by yourself. The only thing you are given are some base classes that you can extend and by doing so, your customized components should work well with Apple's programming paradigm. However, you could get very good results by just customizing the existing components supplied by Apple so in most cases, you should not have to implement and design your own components. One thing to consider if you want to design an interface entirely on your own is that your application will not have the same kind of familiarity as it would have had if you had designed it using standard components. By using standard components, a user with previous iPhone experience can to some extent intuitively know how to use your interface.

Working on the Pomodoro project as a whole has been mostly a smooth process, although problems arose when we had to integrate our application with the cloud. During the entirety of the Project, we have been working at Chalmers while the other teams including the team developing the cloud were working at Logica. This made it very hard to communicate directly with the cloud team which

to some frustration while integrating our application. This highlighted the importance of communication while working on a larger project involving multiple teams.

7. Future Work

This chapter will deal with various aspects of the Pomodoro iPhone project that are not entirely completed or should in some way be looked into in the future.

7.1 Statistics

One of the original requirements of the Pomodoro Application was to be able to view statistics based on the usage of the application. For example, one statistical feature of the application could be to present a graph showing during which hours of the day the user gets interrupted the most or during which hours the user completes the largest amounts of Pomodoros. These types of statistics could give an indication of which time of the day the user is most productive. Unfortunately, there was not enough time to complete this feature and other more highly prioritised features had to be completed instead.

The iPhone SDK does not contain any classes or methods for displaying graphs or charts of any kind so if this feature was to be developed in the future, it would have to be done with some kind of third party framework or an in-house developed solution.

7.2 Offline mode synchronized with online mode

In the application's current state, there is no way of working with information stored in the cloud offline. The offline mode is at least capable of giving the user some way of working offline, but it is not convenient if the user wants to work with information that is stored in the server.

One idea that came up during development was that maybe there could be a way to synchronise offline information with the cloud and vice versa. The main principle behind the idea was that while working towards the web service online, the application should work concurrently towards a local database and store information locally. Should a request to the web service fail, the altered object should be marked as 'not synchronized' in the local database. The next time the user logs in, all objects in the database that are not synchronized with the web service should be committed to the server. This way, information is synchronized between the web service and the database.

There are some complications with this model. First and foremost, there needs to be some way to log in offline and ways to handle accounts offline. Also, there are some scenarios where the above solution will lead to information getting out of synch. One such situation is if the user decides to use another client and modify information in the web service before logging in on the iPhone client. In this case, there is no way of knowing which information is correct and if the iPhone client simply overwrites information according to the above solution, there is a chance that changes that have been made with other clients are overwritten. Should a synchronization feature be developed in the future, these issues need to be looked into before this solution is acceptable.

7.3 Handle OAuth access limit

In the current version of the cloud web service, there is a limit on the number of requests a user can make before he has to log in to the service again. This limit is unfortunately not static, it is specified by the user upon log in. Currently, this access limit is not retrieved or handled in any way, so there is a possibility that the user will run out of the access limit and be unable to use the application before he logs in again. This is something that really needs to be fixed in a future version.

7.4 Improve Plan the Day

As mentioned previously, during user testing, it was discovered that users were unable to understand the advantage of prioritizing activities using the prioritize feature located within the 'Plan the Day' view. This needs to be improved, or maybe the functionality of prioritizing activities should be moved to the 'Select Activity' view so that the result of prioritizing activities is immediately apparent in the same view.

8. REFERENCES

Books

Tidwell, J., 2005. *Designing Interfaces.* United States of America: O'Reilly.

Websites publications

Francesco Cirillo. 2009. *The Pomodoro Technique.* [online] Available at: http://www.pomodorotechnique.com [Accessed 15 March 2010].

Brown, Michael K., 2009. *Which iPhone app is best – Web or native?*. [Online] Available at: http://www.mobilemarketer.com/cms/opinion/columns/3255.html [Accessed 27 February 2010].

Apple Inc., 2009. *Cocoa Design Patterns*. [Online] (Updated 19 November 2009) Available at: http://developer.apple.com/mac/library/DOCUMENTATION/Cocoa/Conceptual/CocoaFundamentals/CocoaDesignPatterns/CocoaDesignPatterns.html [Accessed 17 February 2010].

Nitobi, 2010. *PhoneGap*. [Online] Available at: http://phonegap.com [Accessed 10 March 2010].

iUI, 2010. *iUI*. [Online] Available at: http://code.google.com/p/iui [Accessed 10 March 2010].

iWebkit, 2010. *iWebkit*. [Online] Available at: http://iwebkit.net [Accessed 10 March 2010].

Barney, Lee S., 2009. *QuickConnect*. [Online] Available at: http://quickconnect.sourceforge.net/browser/index.html [Accessed 10 March 2010].

Nitrogen Design, 2009. *LiquidGear iPhone*. [Online] Available at: http://www.liquidgear.net [Accessed 11 March 2010].

Appcelerator Inc., 2010. *Appcelerator Titanium*. [Online] Available at: http://www.appcelerator.com/ [Accessed 11 March 2010].

Mcfadden, Jeff, 2010. *Magic Framework*. [Online] Available at: http://www.jeffmcfadden.com/projects/Magic%20Framework [Accessed 11 March 2010].

Paul, Ryan, 2009. *MonoTouch*. [Online] Available at: http://arstechnica.com/open-source/news/2009/01/open-source-mono-framework-brings-c-to-iphone-and-wii.ars [Accessed 11 March 2010].

The Omni Group, 2010. *Omni Frameworks*. [Online] Available at: http://www.omnigroup.com/developer [Accessed 11 March 2010].

Sapus Tongue, 2010. *Cocos2D*. [Online] Available at: http://code.google.com/p/cocos2d-iphone [Accessed 12 March 2010].

Wikipedia, 2010. *Representational State Transfer.* [Online] Available at: http://en.wikipedia.org/wiki/Representational_State_Transfer [Accessed 22 February 2010].

CapTech Ventures Inc., 2009. *Tutorial: Simple iPhone Rest Clien*t. [Online] Available at: http://blogs.captechventures.com/blog/jack-cox/tutorial-simple-iphone-rest-client [Accessed 22 February 2010].

Wikipedia, 2010. *Simple Object Access Protocol.* [Online] Available at: http://en.wikipedia.org/wiki/SOAP [Accessed 22 February 2010].

OAuth, 2007. *Introduction*. [Online] Available at: http://oauth.net/about [Accessed 23 February 2010].

Wikipedia, 2010. *OpenID.* [Online] Available at: http://en.wikipedia.org/wiki/OpenID [Accessed 23 February 2010].

Nachbaur, Michael, 2009. *How to automate your iPhone app builds with Hudson* . [Online] Available at: http://nachbaur.com/blog/how-to-automate-your-iphone-app-builds-with-hudson [Accessed 10 March 2010].

Zorz, Z., 2009. *iPhone anti-virus software: Has the time come?*. [Online] Available at: http://www.net-security.org/secworld.php?id=8544 [Accessed 21 February 2010].

Images

Fig 1:
Apple Inc., 2009. *Framework object sending a message to its delegate*. [Image] Available at: http://developer.apple.com/mac/library/DOCUMENTATION/Cocoa/Conceptual/CocoaFundamentals/CocoaDesignPatterns/CocoaDesignPatterns.html [Accessed 15 March 2010].

Fig 2:
The very first design of our Pomodoro view.

Fig 3:
Apple Inc., 2009. *Cocoa version of MVC as compound design pattern*. [Image] Available at: http://developer.apple.com/mac/library/DOCUMENTATION/Cocoa/Conceptual/CocoaFundamentals/CocoaDesignPatterns/CocoaDesignPatterns.html [Accessed 15 March 2010].

Fig 4:
Apple Inc., 2008. *Sample Window with Advanced Settings Drawer visible at right.* [Image] Available at: http://developer.apple.com/iphone/library/documentation/DeveloperTools/Conceptual/SharkUserGuide/GettingStartedwithShark/GettingStartedwithShark.html [Accessed 24 March 2010].

Fig 5:
Keychain example

Fig 6:
Apple Inc., 2010. *iPhone*. [Image] Available at: http://www.apple.com/iphone/gallery [Accessed 15 March 2010].

Vodafone, 2010. *HTC Magic (One of many Android models)*. [Image] Available at: http://www.vodafone.com/start/media_relations/news/group_press_releases/2009/vodafone_and_htc_unveil/htc_magic.html [Accessed 24 March 2010].

Fig 7:
Apple Inc., 2010. *Home Screen iPhone*. [Image] Available at: http://www.apple.com/iphone/gallery/ [Accessed 15 March 2010].

Fig 8:
Apple Inc., 2010. *The launch image for the Settings application*. [Image] Available at: http://developer.apple.com/iphone/library/documentation/UserExperience/Conceptual/MobileHIG/IconsImages/IconsImages.html [Accessed 15 March 2010].

Fig 9:
Apple Inc., 2010. *Status bar types.* [Image] Available at: http://developer.apple.com/iphone/library/documentation/UserExperience/Conceptual/MobileHIG/SpecialViews/SpecialViews.html#//apple_ref/doc/uid/TP40006556-CH10-SW1[Accessed 24 March 2010]

Fig 10:
Apple Inc., 2010. *Navigation bar example.* [Image] Available at: http://developer.apple.com/iphone/library/documentation/UserExperience/Conceptual/MobileHIG/SpecialViews/SpecialViews.html#//apple_ref/doc/uid/TP40006556-CH10-SW1[Accessed 24 March 2010]

Fig 11:
Apple Inc., 2010. *Tool bar example.* [Image] Available at: http://developer.apple.com/iphone/library/documentation/UserExperience/Conceptual/MobileHIG/SpecialViews/SpecialViews.html#//apple_ref/doc/uid/TP40006556-CH10-SW1[Accessed 24 March 2010]

Fig 12:
Apple Inc., 2010. *Tab bar example.* [Image] Available at: http://developer.apple.com/iphone/library/documentation/UserExperience/Conceptual/MobileHIG/SpecialViews/SpecialViews.html#//apple_ref/doc/uid/TP40006556-CH10-SW1[Accessed 24 March 2010]

Fig 13:
Apple Inc., 2010. *A navigation bar can contain a navigational control*. [Image] Available at: http://developer.apple.com/iphone/library/documentation/UserExperience/Conceptual/MobileHIG/SpecialViews/SpecialViews.html#//apple_ref/doc/uid/TP40006556-CH10-SW1[Accessed 16 March 2010].

Fig 14:
Apple Inc., 2010. *A multi-segment back button is not recommended.* [Image] Available at: http://developer.apple.com/iphone/library/documentation/UserExperience/Conceptual/MobileHIG/SpecialViews/SpecialViews.html#//apple_ref/doc/uid/TP40006556-CH10-SW1[Accessed 16 March 2010].

Fig 15:
Tab bar with the main options

Fig 16:
Tab bar with icons

Fig 17:
First window under the Settings tab. Pushing ‘Pomodoro Settings drills’ the user down one window. Pushing ‘Pomodoro Length (minutes)’ drills the user down a second time, reaching the end screen

Fig 18:
Start screen

Fig 19:
Step by step adding an activity

Fig 20:
Detail view of an activity

Fig 21:
Select activity screen

Fig 22:
Pomodoro timer for activity and break

Fig 23:
Shows how to add interruptions and how the segment control changes

Fig 24:
Pomodoro has finished

Fig 25:
Plan the day

Fig 26:
Prioritise activities in the current To Do

Fig 27:
Settings

Appendix

1. Time Plan

Feb 1 - Feb 28

- Evaluation of technologies and interface design practices on the iPhone.

March 1 - March 10

- Evaluation of technologies contd. and begin design of interface prototypes according to best practices.

March 10 - April 4

- Set up an Integrated Development Environment (IDE).
- Perform proof of concept to make sure that it is possible to integrate all necessary functionality in the Pomodoro project.
- Perform user testing on the different interface prototypes completed earlier in March. User testing will be performed according the design method *paper prototyping* to get early feedback from the design.

April 5 - April 11

- Implement a skeleton interface for the whole application.
- Implement the functionality necessary to perform a Pomodoro.

April 12 - April 18

- Implement CRUD for the activities (offline).

April 19 - April 25

- Implement a settings menu with relevant functionality.

April 26 - May 2

- Connect to the cloud, REST & JSON - CRUD Activity and Start/Stop Pomodoro.

May 3 - May 9

- Implement interruptions internal/external and plan the day (without estimations) functionality.

May 10 –

- Implement the statistic feature: statistics of the interruptions (internal/external) during a day based on all data from the account.
- Estimations and records.
- Plan the day with estimations.
- Make a more complex activity list with full functionality.

- Make the settings menu more complex.
- Use extended services from the cloud.
- Interruptions, adding urgent activities.

2. Requirements Supplied by Logica

Feature Requirements:

Execute a Pomodoro

1. Define Activity for current Pomodoro.
2. Start the Pomodoro timer, default 25 minutes.
3. Alert user when Pomodoro timer reaches zero.
4. Stop the Pomodoro timer, a break count up starts.
5. Mark Activity as done.
6. Archive Activity when marked as done.
7. Inform user every four Pomodoros for a longer break.

Activity Lists

8. CRUD (Create, Update, Delete) Activity to To Do list.
9. CRUD Activity to Activity Inventory.
10. CRUD Activity to Unplanned & Urgent list.
11. View To Do list.
12. View Activity Inventory list.
13. View Unplanned & Urgent list.
14. Move Activity from To Do list to Activity Inventory.
15. Move Activity from Activity Inventory to To Do.
16. Move Activity from Unplanned & Urgent to either To Do or Activity Inventory.
17. Sort on columns for To Do and Activity Inventory.

Connect to cloud

18. Authenticate user with the cloud.
19. CRUD operation in the cloud.
20. Create and read Pomodoro Statistics in the cloud.

Records and estimations

21. Set estimated Pomodoros for an Activity.
22. Track number of Pomodoros executed for an Activity.

Interruptions

23. Note an Internal Interruption, increase value for the current Pomodoro.
24. Not an External Interruption, increase value for the current Pomodoro.
25. Create Activity to Unplanned & Urgent list.

Settings

26. Change the length of a Pomodoro.
27. Change the length of a break between Pomodoros.
28. Change the number of Pomodoros until a longer break.

29. Change the cloud domain/URL.
30. Change login information for the cloud service.

Statistics

31. The user can view statistics based on his or hers usage of the application, see Appendix 4.

Plan the day

32. Define number of Pomodoros planned to be executed today.
33. Move Activities from Activity Inventory list to the To Do list.
34. Inform the relation between planned executed Pomodoros and total estimation of Activities in the To Do list.

Non-Functional Requirements

1. The application should support several languages with a minimum of two where English is mandatory.
2. The design must be based on the domain model.
3. Separation of concerns should be applied in the design.
4. User interface should contain the requested views, see Appendix 3.

3. Views

The user interface and graphical design is not defined in detail to allow each platform to leverage their benefits. To make a cohesive user experience between the applications, there are a set of views defined that should always be implemented.

1. **Pomodoro View**
   Contains the Pomodoro timer and any actions related to the execution of a Pomodoro.
2. **Activity Lists**
   The Activities should be listed in separate views. The different lists could be displayed using sub pages or filter functionality.
3. **Settings**
   Changing settings should be a separate view.
4. **Statistics**
   All statistics should be presented in a separate view.
5. **Plan the day**
   A special view should be used when planning today's activities.

4. Statistics

1. Number of Pomodoros executed.
2. For each Pomodoro
   a. Number of internal interruptions
   b. Number of external interruptions
   c. Start time
   d. End time
3. For each day
   a. Number of Pomodoros executed
   b. Number of Activities completed
   c. Number of internal interruptions

d. Number of external interruptions

4. Interruptions
    a. Show a graph when internal interruptions occur during the day
    b. Show a graph when external interruptions occur during the day
    c. Show a graph when interruptions occur during the week
    d. Show a graph when interruptions occur during the last 30 days, scale 1 day

5. Original Domain Model Supplied by Logica