FINAL REPORT:

MOBILE JOB LIFECYCLE AND PARTS MANAGEMENT APPLICATION

Author: Daniel J Workman
Supervisor: Dr. Andrew Jones
Moderator: Dr. Xianfang Sun

Table of Contents

Introduction **4**
Interim Report Feedback *4*
Specification and Design **4**
Updated Functional Requirements *5*
User Interface *5*
Web Service *6*
User Tracking *6*
Data Considerations *7*
Reading and Writing 7
Quantity of Data 8
Implementation **9**
Web Service *9*
Application Framework *10*
Communication Modules *10*
HTTP Requests 10
JSON 11
Progress Indicator 11
User Tracking *11*
Barcode Scanning *12*
Issues Encountered *13*
Results and Evaluation **14**
Testing *14*
Usability Tests 15
Fault Testing 15
Performance Testing 15
Security Testing 16
Client Testing and Feedback *16*
Design Process *17*
Future Work **17**
Security Improvements *17*
Improvements to Data Storage *18*
Extra Features *18*
Re-factor Inelegant Solutions *18*
Communications Module 18
Web Service 19
Extraction of Business Logic from the UI 19
Conclusions **19**
Reflection **20**
Glossary and Abbreviations **21**

References.....**22**

Introduction

In the period between the interim report and completion of the project a number of changes have been made which require highlighting.

Firstly the client and I discussed some minor changes to the requirements, which have been discussed in the Design section. Although the project began by following the agreed AGILE approach a loss of project code during the implementation phase caused some delays and a somewhat modified development process. This will be discussed in detail later in the project report.

Other than this set back the client required an application that was much the same as was originally agreed and the design phase began almost immediately following the interim report.

Interim Report Feedback

After reading the feedback provided by both the supervisor and moderator it was decided that there were two main points that should be addressed:

Firstly, it was noted by the supervisor that the interim report lacked an architecture diagram, which would have given a better idea of how the system interacted as a whole and how data flowed between each area. I have therefore included an appropriate architecture diagram, which is discussed in the Design section of this report.

Secondly, the moderator suggested that changing the language of the web service from C# to a better-known language would overcome the problems caused by a lack of intimate knowledge of said language. After some discussion with the client it was agreed that this would not be an issue and I made the decision to move to a PHP and MySQL implementation. These are both technologies that I have used before and am comfortable developing in and should therefore speed up the implementation process quite considerably. Again, this is covered in more detail during the Design section of this document.

Specification and Design

This stage of the development was approached in the following manner:

- After liaising with the client minor updates were made to the requirements.
- A basic user interface design was created via storyboarding whilst inspecting the requirements and system flow diagram.
- The technologies behind web services were considered and selected.
- A system for storing and logging in users was considered.
- How the system should interact with data was considered and a solution found.

Updated Functional Requirements

The following are the functional requirements provided by the client and covered previously in the interim report. They have been reformatted in a more technical and concise manner and include some minor updates, which were not originally stated at the beginning of the project.

1. The solution shall be designed with a job lifecycle centric workflow.
    1.1. The solution shall be able to accept, complete and abort jobs.
        1.1.1. The solution shall be able to view all waiting jobs (for a specific engineer).
        1.1.2. When a job has been accepted all waiting jobs shall be made unavailable.
        1.1.3. When aborting a job a valid reason shall be provided.
2. The solution shall be able to view the details of a job (waiting or accepted).
    2.1. A job contains details pertaining to the job as a whole, the company (client) the job is for and the specific ATM the company has/needs.
    2.2. Job details shall include the job name, number and type.
    2.3. Company details shall include name, address, and a contact number.
    2.4. ATM details shall include the make, model and any relevant notes.
3. The solution shall be able to add parts to an accepted job.
    3.1. The solution shall be able to scan part barcodes.
        3.1.1. Barcodes shall use the Code 128 symbology.
    3.2. The solution shall allow for manual insertion of part numbers.

It could be argued that there are very few functional requirements provided by the client for this project. However, considering the scope of the project and its intended final use (an internal tool rather than a marketable product) the client was not overly concerned with the look and feel of the application but rather that it could meet the needs of the engineer in their day-to-day work flow. This is therefore a good example of a project that prioritises function over form.

The additional features requested by the client remain as before and a summarised version has been reiterated below:

- Provide extra information with each job such as:
    - Parking details
    - A ‘street view’ map showing the location of the site.
    - An area where job specific documents can be viewed (most likely PDF files).
- A list of all completed jobs, for the specific engineer and for that day only.

User Interface

Using both these requirements and the provided system flow diagram I went about mocking up a basic UI design at the earliest opportunity. I began by creating some quick sketches, which showed the general flow of the application from view to view and the UI elements one would expect for this style of application. The client liked both the look and feel of the application and the layout of the UI

elements so I took this feedback and decided to storyboard the application using XCode's built in storyboarding feature.

The storyboarding feature in XCode was introduced relatively recently in version 4.2 (released October 12, 2011) and allows for very quick creation of GUI's and also automatically creates code to control navigation and events for each view. This allows the user to create a multi-view application that can be fully navigated before even writing any code. This amalgamation of XCode's existing Interface Builder with automated code generation was invaluable when designing the interface as it highlighted design issues long before any code need have been written. I was therefore able to go through multiple revisions of UI design without having to scrap or modify any code. It also allowed the client to get a good idea of how the application would operate and navigate with little effort on my part. Some lightweight dummy data was also hardcoded into specific views such that the client could even select an example job and view its individual sections, again with no written code.

A screen shot of the final storyboard layout has been included in Appendix B. The arrowed links between each view demonstrate a 'segue' (a transition from one view to another). The left most view named the 'Navigation Controller' provides each subsequent view with inherited navigation controls so that 'Back' buttons are automatically generated and need not be coded individually.

Web Service

As stated in the interim report my main priority was to get a working implementation of the web service up and running as soon as possible. The decision to move to PHP from C# meant that most of my initial research toward a C# implementation was no longer of use. My knowledge of PHP however was much greater than that of C# so this did not provide a significant set back.

Before thinking about language specific details it was important to decide what type of web service should be used. There exist two main routes one can take when considering implementing a web service. The well structured and strongly typed SOAP protocol and the more lightweight RESTful approach. Having studied both approaches only this year I already had a good idea of how each method worked and therefore did not feel the need to perform much background research on either. It is my opinion that creating a RESTful API is both simpler and quicker to implement rather than the somewhat bulky and potentially over-engineered approach provided by SOAP. Taking into account the delays that the web service had already incurred the obvious choice to me was to take the RESTful approach.

The next decision was to select an appropriate relational database to store my application data. Although the client utilises a Microsoft SQL Server database it made more sense to develop using readily available databases that would require no set-up cost. The school hosts both Oracle and MySQL implementations for which I already have credentials for each. I therefore made the choice to use the MySQL implementation as it is the one with which I am most familiar.

User Tracking

I realised very soon as I began the implementation phase that I had overlooked an important area of the application. Within my initial design phase I had not considered how different users (engineers) were to be differentiated within the application.

I initially saw two methods of achieving this:

- Store the engineer's identifier with each new installation of the application (thereby hardcoding each instance of an installation to a specific engineer).
- Implement a login feature such that any engineer might use any device that had an instance of the application.

Considering each engineer were provided with their own device the first method would seem an adequate method of achieving the required functionality. It would also require much less effort to implement than a full user tracking system. After discussing this with the client however they felt that it would not be unreasonable for an engineer to either borrow another device in the event of their own devices failure or to use another engineer's device if two engineers were assigned to the same job. It was therefore decided that a method of logging a user into the system would be required.

The main storyboard was updated to include an initial login screen, which would prompt the user for their credentials (a username and password) and the database schema ensured the engineer table stored the credentials in the correct manner (a password hash and salt rather than plaintext passwords).

Data Considerations

It is important we understand at an early stage how data flows through the system. Using the architecture diagram included in Appendix A, we can get a basic idea of how both the workflow and data flow should be achieved.

The left side of the diagram shows a simplified view of the equipment in use at the client's office. One or more employees, based at the office, create and modify jobs using fixed terminals. These jobs are then stored in the client's relational database.

The right sight of the diagram illustrates one or more field engineers with smartphones who are communicating via the Internet with the web service, which is hosted on the client's server. The web service then communicates with the database to return or modify any data the mobile application has requested.

Although the interim report specified a Microsoft SQL Server database I will be using a MySQL implementation to develop this application. Considering the access of data is abstracted by the web service a change in database implementation would not present a major problem. The application will never need to directly interact with a database so all that would require modification is the web service itself.

Reading and Writing

Of interest is the balance between the reading of data and the writing of data. By inspecting the requirements we can see that the majority of data interactions within the application are that of the reading and display of data only. There are only two specific times that the application may write any data back to the database:

- When adding parts to a job.
- When changing the status of a job (accepted, completed or aborted).

This provides some insight into how we might structure the communication elements of the project. Had the application needed to write large amounts of data or done so quite frequently then one method of improving efficiency could be to store up a number of write requests to reduce the amount of times the application sends a request to the web service. Considering how little time the application should be sending write requests however, I feel that it would be adequate to immediately send write requests as and when they are generated by the application.

Quantity of Data

Another important consideration is the amount of data we are expecting to receive and transmit. If the application were to require the transmission of large amounts of data such as audio or video we would need to think carefully about how frequently we communicate with the database and whether we need to cache data to reduce the time spent downloading that which could have been stored in memory from a previous communication session.

Data that could potentially benefit from being cached might include:

- An engineer's waiting jobs.
- An engineer's current job.
- An engineer's completed jobs.
- A list of all recognised parts.

Considering the purpose of the application we know that we need only be sending and receiving textual and numeric data in nearly all cases. This should mean that all responses and requests from and to the web service should be relatively small in size and therefore not require considerable bandwidth. This is extremely relevant when we take into consideration that the data rate achieved by mobile phones can vary greatly due to variances in cell signal strength and also the supported data service of the currently utilised Base Transceiver Station (cell tower). Older stations may only support the GPRS service whereas newer ones the more modern third or even fourth generation services.

As a result I believe that it should not be necessary to a design a system whereby data is stored temporarily as a method of reducing overall application bandwidth.

There are two main advantages to this methodology:

- We do not need to worry about cached data that has become invalid between communication windows (e.g. a waiting job has been removed by head office yet it still appears within the application).
- The sending and receiving of data is highly simplified as we need only communicate with the web service as and when the data is required.

There is however one instance where the application would be required to download (potentially) large quantities of binary data. The "Associated Documents" feature, which the client has requested as a potential extra feature and is not a requirement, would mean that the application might need to

download PDF data in certain cases. I feel however that this feature should not denote the overall design of the project. If the feature were to be implemented then a standalone method of caching for these documents could likely be implemented without considerable effort.

Implementation

Once I was happy that enough consideration had been given toward the overall design I began the implementation stage of the project. I felt that the important design considerations that denoted fundamental design decisions were complete and that any additional design problems could most likely be finalised as I learnt more about how I could achieve my goals using the tools available.

The implementation phase involved the following events in priority order:

- Implementation of a working web service.
- Creation of a basic multi-view iOS application using the storyboard as a guide.
    - Creation of a module that could communicate with the web service via HTTP.
    - Implementation of user tracking (logging in of specific users).
    - Use of the two previous features to pull job data from the server and display it within the app as appropriate.
    - Implementation of the acceptance, completion and abortion of jobs.
    - Implementation of the scanning feature for an accepted job.
    - Implementation of the extra features the client requested.

Web Service

I began by ensuring I still had access to the school's MySQL database. The location of the database can be resolved using the following URL: *ephesus.cs.cf.ac.uk,* assuming the use of a suitable database client and valid credentials.

Once access had been confirmed I liaised with the client so as to agree on a valid schema for the database. I began by creating a basic SQL schema that would provide storage for all data that had been agreed as a minimum requirement and also made sure that the table structures and data types were compatible with the client's database. This would allow for a much smoother integration once the client wished to run the application using live data from their own database. The SQL script was designed such that it could be run repeatedly without conflicting with the existing tables in the database. The script first deleted all tables (if they existed), then recreated them and finally seeded them with dummy data for testing purposes.

As I had never implemented a web service myself and had only studied them theoretically I performed some basic research so as to find a suitable starting point. I soon came across a very good tutorial that provided a basic framework from which a RESTful API could be implemented using both PHP and JSON [1].

I would need somewhere to host my web service and I looked into the options available to me. Considering the school provided server space to students this seemed like the most logical choice rather than to either run my own server from home or to pay for server space from a commercial

source. The web service was therefore hosted on the schools *users* server, which can be found at *users.cs.cf.ac.uk.*

Once I had somewhere to host my service I moved on to the implementation. As PHP includes bindings for MySQL, connecting to my database and accessing the required data was achieved with ease. I implemented the service such that requests could be sent using the POST data included in the HTTP request. Conditional statements checked for the requested functionality and return the requested data or a message denoting success or failure. Any data that is to be returned is encoded using JSON such that it can be unpacked on retrieval by the application.

Both the web service and accompanying database schema have been added to a ZIP archive and will be available with the submission of this report.

Application Framework

As stated previously in the Design section XCode's storyboarding feature had already created a basic and navigable application without the need for any user-generated code. To begin customising each view however 'view controller' classes needed to be created. I therefore went about creating individual controller classes for each view in the application. Once this was complete I was able to make programmatic changes to each view that were not possible using the static nature of the storyboarding tool.

I then went on to create a basic Constants header file that could be used to store any relevant constants the application may need as the project evolved.

Communication Modules

HTTP Requests

To successfully communicate with the web service the application must be able to send and receive HTTP requests. Although iOS provides full support for creating, sending and receiving such requests the built in tools are somewhat bulky to use and require a relatively large amount of configuration before each request can be sent [2]. The tutorial [3] used during the creation of the web service recommended the use of a library called ASIHTTPRequest [4], which greatly simplifies the process and allows for requests that can be created and sent in only a few lines of code:

```
ASIFormDataRequest *request = [ASIFormDataRequest requestWithURL:[NSURL URLWithString:urlString]];
[request setPostValue:@"login" forKey:@"request"];
[request setPostValue:username.text forKey:@"username"];
[request setDelegate:self];
[request startAsynchronous];
```

The first line takes the destination URL (in our case the location of the web service) and creates a new request object that is ready to accept POST data. We then set the POST data values (in this instance we want the credentials for a specific user so we may log them in) and finally we set the delegate to the current class (so that we can catch the request response) and then send the request asynchronously.

The data the web service returns must be encoded in an appropriate format such as XML or JSON. XML's inherent verbosity (the need for closing tags for example) makes for a somewhat larger data footprint when compared with JSON, which has a considerably simpler notation. My personal

preference would be to use JSON as it is easy to parse and, from my own experience, appears to be the more favoured approach when considering web services.

JSON

The JSON library suggested by the tutorial is SBJson [5]. An example from my project of the library in use is as follows:

```
NSString *responseString = [request responseString];
NSDictionary *responseDict = [responseString JSONValue];
NSString *hash = [responseDict objectForKey:@"pw_hash"];
NSString *salt = [responseDict objectForKey:@"pw_salt"];
```

First we take the response string as returned by the web service (text containing JSON encoded data) and we call the *JSONValue* procedure on our string. This returns us a dictionary of key-value pairs, which can be queried with ease.

Progress Indicator

A problem associated with asynchronous communication is that the user has no indication that the application is performing the task. The web service tutorial recommended the use of a library called MBProgressHUD [6], which solves this issue by displaying a throbber in the centre of the users display and a suitable message so that users a) know that they must wait until we have received a response and b) cannot use the application until the action is complete. The library was created because although the OS does have a similar feature its API is both private (Apple do not allow its use in commercial applications) and undocumented [7].

The progress view can be created with the following lines of code:

```
MBProgressHUD *hud = [MBProgressHUD showHUDAddedTo:self.view animated:YES];
hud.labelText = @"Logging in...";
```

And dismissed with the following:

```
[MBProgressHUD hideHUDForView:self.view animated:YES];
```

Screenshots showing the feature in operation can be found in Appendix C.

User Tracking

To store credentials securely we must always store them as a hash of the original plaintext password, which has been salted with a suitably large and unique salt. We must also select a cryptographic algorithm that is strong enough to resist attacks from methods such as brute-force and rainbow tables.

As one would expect, we are provided with cryptographic functions within iOS, which can be used for this very purpose. I did, however, elect not to use these built-in tools. I did so for the following reasons:

- The built-in tools do not provide an implementation of an “adaptive” algorithm.
- There are no built-in salt generation tools.

An adaptive cryptographic function provides the ability to slow the work factor by increasing the number of iterations required. This renders brute-force attacks unfeasible, as each encryption cycle can take considerably longer to perform than other popular non-adaptive algorithms.

An algorithm that satisfies these requirements is bcrypt [8], which makes use of the Blowfish cipher and allows for a variable number of rounds as defined by the user. An implementation for Objective-C exists by the name of JFBCrypt [9], which I incorporated to my project.

The following two lines illustrate the generation of a random salt for the user (including the number of rounds we wish the hash function to use) and the hashing of the actual password and salt:

```
NSString *salt = [JFBCrypt generateSaltWithNumberOfRounds: 10];
NSString *newhash = [JFBCrypt hashPassword: password.text withSalt: salt];
```

This newly generated password hash and accompanying salt are then stored in the database and read whenever a user tries logging into the system.

Barcode Scanning

As stated in the interim report the selected library for barcode scanning was ZBar [10]. Having already created a 'proof of concept' application during the early stages of the project I was already familiar with the library's API and how to go about integrating it with my own application.

The scanner operates by creating a pop-up modal view, which shows a live feed from the camera. A message is sent to the delegate class when a code has been found so that we may handle the response. The following code achieves this:

```
ZBarReaderViewController *reader = [ZBarReaderViewController new];
reader.readerDelegate = self;

// Disable all other symbologies
[reader.scanner setSymbology: 0
                      config: ZBAR_CFG_ENABLE
                          to: 0];

// Enable Code 128 only
[reader.scanner setSymbology: ZBAR_CODE128
                      config: ZBAR_CFG_ENABLE
                          to: 1];

reader.readerView.zoom = 1.0;

[self presentViewController:reader animated:YES completion:nil];
```

We begin by creating a reader object and setting our current class as the delegate (so that we may catch any messages the reader returns). By default the reader will recognise all types of barcode it provides support for. As we only care about barcodes that conform to the Code 128 standard we first disable all symbologies and then enable only the Code 128 symbology.

Lastly we ensure the cameras zoom level is set to the lowest possible value to give us the greatest possible viewable area and display the scanner view to the user.

We also implement a handler function called *didFinishPickingMediaWithInfo,* which will fire once the reader finds one or more barcodes within the target image.

The contents of this function are as follows:

```
[reader dismissViewControllerAnimated:YES completion:NO];

id<NSFastEnumeration> results = [info objectForKey: ZBarReaderControllerResults];

ZBarSymbol *symbol = nil;
for (symbol in results) {
    NSInteger partId = [[symbol.data stringByTrimmingCharactersInSet:[NSCharacterSet
characterSetWithCharactersInString:@"!A"]] integerValue];

    partIdField.text = [NSString stringWithFormat:@"%d", partId];

    // Turn on the add button as we have scanned a code
    self.navigationItem.rightBarButtonItem = addButton;

    // We don't care if the scanner finds multiple codes as the engineer should only scan one code
    break;
}
```

Here is a summarised list of what we are achieving with the above code:

- Dismissal of the scanner view as it is no longer required
- Retrieval of a list of barcodes the scanner was able to recognise.
- For the first code we take the barcode data and strip the preceding “!A” encoding. This is done because part numbers are not stored in this format in the database.
- We set the UI element that holds the part number to our scanned part number.
- We enable the “Add” button on the view so the user may add the part to the job.
- We break out of the loop, as we are only ever interested in one barcode.

Issues Encountered

During the course of the implementation period there were a number of issues, which affected both project timescales and the quality of the overall application.

By far the greatest problem encountered during development was the steep learning curve presented by Objective-C, a language that I had previously not used or studied. Although I was quite familiar with other high-level object-oriented languages such as Java and C++, Objective-C has quite a different approach toward object-oriented programming and a syntax that differed enough to cause frequent confusion. The on-going learning process of both the language and the tools provided meant that the initial implementation phase of the application began much slower than I had anticipated.

Fortunately these delays were alleviated by a fantastic wealth of knowledge that could be found from either Apple’s own documentation site or sites such as *stackoverflow.com*. I found that many other developers had experienced similar problems and as a result good quality and professional resolutions could usually be found. This method of learning ensured that although slow to begin, the project remained, for the most part, on schedule.

Another serious and often aggravating set back came about from Apple’s strict certification process. Although XCode comes bundled with a good device emulator it is always necessary to test software on a physical device. To do this Apple requires the user to register any devices they wish to use for testing within their developer portal. Certificates must also be issued for any computer used for development. Even when these requirements had been met XCode would often give rather cryptic

error messages relating to invalid certification or insufficient user rights. The problems were also compounded by the fact that being a member of an organisation meant I was unable to interact directly with the developer portal and would need to liaise with Rob Davies who had the administrator privileges required to make the changes I required. This problem was finally resolved as I was fortunate enough to know another iOS developer who had his own personal developer account with Apple. I was therefore finally able to interact with the portal directly and ensure all the correct certificates were in place. It did mean however that for the first month of development I was only able to use the emulator to test my code. This becomes even more relevant when we consider that the emulator cannot provide testing of the scanning feature, as a physical camera is required to do this.

During the process I did not find that any of the libraries I was using were lacking in documentation or features. Each and every library I used integrated seamlessly with my project and interfacing with them was extremely simple. I was fortunate therefore not to incur any project delays due to issues that were potentially out of my control without switching to alternative libraries or by implementing my own solutions.

The final significant issue encountered was that of the loss of much of my project code due to the theft of my laptop. Although considerable amounts of my project were backed up I lost what was, conservatively, a months worth of coding, which would therefore need repeating. When I returned to the coding process I was fortunate to have a better understanding of the language and could get back to a similar stage in considerably less time. I was however quite worried about the reduced timescales I was going to have to face and as a result took some shortcuts and poor design choices which I would not have made had I had a longer period of time to perform the implementation. Of more importance to myself was that although the design was not as elegant as it could be I was aware of the shortcomings I had introduced and could therefore address them in the future if necessary. I will therefore cover these issues in the Evaluation section of this report.

Results and Evaluation

At the point at which the implementation phase came to an end the project included all of the required features as well as the majority of the additional features. I had found during the implementation that the extra features requested did not require much effort to complete and I had therefore found the time to complete them as I was writing the other features.

Once the implementation phase was complete I began to think about the formal testing phase of the application. Although I had been testing during the implementation phase there were certain extra tests that were carried out once the application was compete. I have covered all the tests performed and the results of these in the following sections.

Testing

I performed extensive testing of the application both during the implementation process and once the required feature set had been implemented. Once I felt that my own testing phase had been adequately completed I provided the client with a copy of the application such that they may

perform there own testing which would undoubtedly highlight problems or oversights that I had not considered during my own testing phase.

Usability Tests

The earliest testing I performed was to ascertain how well the UI layout had been designed. Before the implementation of any of the major features I provided a few people who had no prior knowledge of the system or its purpose with the basic navigable UI and observed how they interacted with it. I also asked them questions about whether they could work out what the application was trying to achieve and whether the navigation system and general flow were logical.

In general it could be seen that the design was straightforward and logical to follow but it was noted that the flow from waiting jobs to accepted jobs was a little confusing. Although I initially considered changing the design of the home screen in an attempt to better illustrate the flow of jobs through the system I inevitably decided against this. The reason for this is that it must be remembered that this application has been designed for internal use only and is not something that would be released to the wider world. Our intended users (engineers) are those who are already fully aware of how the business operates and should therefore understand the flow of the application without further instruction. Even if an engineer were to misunderstand an area or feature of the application it would be possible to provide them a small amount of training in the use of the app as we have direct contact with our application users.

Fault Testing

Once the application could successfully communicate with the web service I began to notice that in certain cases the application was not reporting errors to the user correctly. To fully test the application's error reporting code I constructed web service requests that were intentionally incorrect or malformed. Using this method I was able to trigger accurate error responses that would be the same as those encountered when the application was live.

The first stage of this testing phase was to manually send POST requests to the web service from outside the application. This tested the response of the web service rather than how the application dealt with those responses. To do this I made use of a tool called curl [11]. curl is a command line tool that allows the user to create and send HTTP requests with ease. The following is an example usage of the tool and the corresponding response from the web service:

```
> curl --data "request=part&part_id=123456" users.cs.cf.ac.uk/D.J.Workman/
Part id <123456> does not exist
```

The second stage was to test the application's handling of these errors as reported by the web service. This test was somewhat slower to complete, as it required modifications to the coded requests within the application rather than the speedy approach provided by curl. It was however an extremely worthwhile test as I found a number of examples whereby the application would either send no error message whatsoever or a generic error message, which would not be very helpful to potential users. Examples of error handling responses can be seen in Appendix C.

Performance Testing

As highlighted earlier in the report the application will have to perform well in a range of differing signal strengths and data schemes. To ensure the application would perform well in all realistically expected situations two performance tests were undertaken. The first was carried out using the

device emulator and not on a physical device. A tool provided by Apple called the Network Link Conditioner allowed for artificial throttling of the computer's Internet connection speed to emulate different mobile data schemes. I tested the application using schemes that would likely be encountered in the field such as GPRS, EDGE and 3G. Even testing at the slowest throughputs expected (approximately 60kbps as provided by a poor GPRS connection) the application still operated within acceptable timescales with no download or upload taking more than about 5 or 6 seconds.

Once I was able to test the application on a physical device I performed similar tests by forcing the device to use slower data schemes (3G can be disabled in the device's settings) and was able to see similar results. In all tests the application communicated well within acceptable limits.

Security Testing

It was noticed during the implementation stage that the security of the web service was very poor. By transmitting data using insecure HTTP requests the service is vulnerable to unauthorised access whether via eavesdropping or by directly accessing the service itself. The lack of authentication of which user is making the requests also means it is very easy to modify the contents of the database without being a valid user.

This problem was discussed with the client but considering that this is only a test database they did not feel time should be spent securing a web service that would inevitably change before the project is made available to live data. Considering also the overheads involved with buying valid security certificates to allow for secure transmission it was decided that this should not enter the scope of the project. As a result of this decision no further security tests were undertaken, as it was well known that the service was insecure and testing would therefore not be required.

This problem has been covered in the Future Work section of the report.

Client Testing and Feedback

Once the application was deemed to be of an acceptable quality and had been tested thoroughly a version was supplied to the client so that they may perform their own tests. This section of the report does not go into detail about the specific tests the client performed but on the feedback they provided as a result of their testing.

The client reported that they were happy with the general look and feel of application and that the method in which the application interacted with jobs was adequate. They were also very impressed with the scanning feature, which was able to accurately recognise barcodes in nearly all test cases. This came as no surprise as the scanning library had performed well during the early prototyping stage. The client used the test database I had myself been using for database. This was done because the client was not happy opening the application to live data until they were happy that the application was secure and provided their required functionality.

There was however an important piece of feedback which could potentially undermine the fundamental design of the application. The client informed me that it was common for engineers to work in conditions where no mobile phone signal was available whatsoever. The application would therefore be unable to work reliably as requests are always sent immediately as the user makes

them. The client recommended that the application have a feature whereby modifications to jobs (mainly the adding of parts as this is what we would expect an engineer to be doing once at the site) could be stored in memory and then sent back to the database once reception returns. The fact that this advice came so late in the development cycle meant that it was not feasible to include this before the project's end and it is therefore covered in the Future Work section of this report.

Other than this problem the client was generally happy with all other aspects of the application.

Design Process

Although the interim report had stated that my intention had been to follow an AGILE method of development a number of issues during the project meant that the project did not manage to follow this methodology effectively. After the loss of the initial project codebase there was considerably less time to implement the project than before. As a result much of the features were implemented without the frequent client contact you would expect from an AGILE approach. The fact that the client was based quite some distance away also made it difficult to demonstrate features effectively without taking considerable time out to travel and meet. The project therefore took more of a traditional Waterfall method whereby features were implemented and then tested once complete with a final deliverable which had had little input from the client during the implementation process. Had the project not suffered the delays it had I am sure that more input would have been possible and the issue raised by the client could have been addressed before the project came to an end.

Future Work

Considering the problems and shortcomings illustrated during the projects evaluation I have identified four main areas in which the project could benefit from additional work.

Security Improvements

Of high importance is the security between the web service and the application. The following steps should be taken to ensure the web service is secured against unauthorised access.

By using the HTTPS protocol rather than simply HTTP alone we would be able to create a secure channel of communication in which to send and receive our requests. This would protect against eavesdropping, which an attacker may utilise to gain sensitive information such as a user's password when they are attempting to authenticate with the web service. Security certificates would need to be acquired (ideally from a recognised Certificate Authority) which would likely carry a cost overhead. This factor contributed to the initial decision not to implement a secure service.

The second important stage of the security improvements would be to authenticate the user for every web service request they make. At present the only authentication that is performed is when a user is attempting to log in to the application itself. No authentication is performed at the request level and as such is vulnerable to modifications that do not originate from a specific user whatsoever.

To remedy this problem the user's credentials (encrypted due to the HTTPS protocol) should be sent with each request. An error could then be returned if the credentials do not match those that are stored within the database.

Improvements to Data Storage

As highlighted by the client the application does not allow for the use of the application when reception is unavailable. To remedy this issue in-memory containers for job data should be implemented such that the application can use this data when an active data connection is unavailable. When reception does become available these in-memory containers can be refreshed and waiting updates can be sent.

Specific Job and Part classes could be created which contain lists of job and part items. The "Current Job" view could incorporate a "Save" or "Update" button, which the user could press when they notice they have regained mobile reception. The functionality could also be triggered automatically when a valid data connection is recognised by the application itself.

Extra Features

The majority of the additional features the client requested were incorporated during the implementation stage. The only feature that was missed was the "Associated Documents" area. The effort to implement did not fit the timescale available and was therefore not included in the final deliverable.

The files themselves should be stored on the client's server but not within the database itself. Storing large amounts of BLOB data in a relational database is inefficient so instead the database should store a URL, which points to the files location on the server.

The application should use the built-in file download tools to download the PDF data and store it in a "documents" list associated with the Job class as discussed in the previous feature. Considering that the files may have the potential to be quite large (in the order of megabytes) they should only be downloaded if the user explicitly requests to view one. The document should then be stored in-memory so that the user can view it multiple times without the need to re-download. On completion of the job the memory could be released, as the document would no longer be required.

Re-factor Inelegant Solutions

Due to the implementation delays a number of shortcuts and inelegant solutions were made when implementing the application. Although the features work the code should be re-factored so as to be neater, more efficient and less likely to cause issues in the future.

Communications Module

The ASIHTTPRequest library would benefit from sub-classing so that we may add our own specialisations in one centralised location. At present these two lines appear anywhere that we are performing a web service request:

```
NSString *urlString = [[NSUserDefaults standardUserDefaults] stringForKey:@"database_url"];
ASIFormDataRequest *request = [ASIFormDataRequest requestWithURL:[NSURL URLWithString:urlString]];
```

First we check the applications settings to find the currently set database location, we then use this database URL to create a request object.

This could be reduced to a single line if we created our own request class, which automatically checked the database URL before each request. The reduced code could then be:

```
ASIFormDataRequest *request = [MyRequestClass getRequest];
```

Web Service

The web service is at present one large file and differing functionality is acquired using a 'request' string, which is provided with the POST data. Although this works it is inelegant and makes the PHP code difficult to read and modify.

By splitting the web service code into separate files and placing them under different directories we end up with a much neater solution and an easier service to modify when necessary. Instead of sending all requests to our root URL (in this case *users.cs.cf.ac.uk/D.J.Workman/*) the structure could be split into logical directory structures such as:

- *users.cs.cf.ac.uk/D.J.Workman/user*
- *users.cs.cf.ac.uk/D.J.Workman/jobs*
- *users.cs.cf.ac.uk/D.J.Workman/jobs/job*
- *users.cs.cf.ac.uk/D.J.Workman/part*

This would remove the need for a request string and create PHP files that are considerably more human readable.

Extraction of Business Logic from the UI

The present solution blends UI code with the business logic of the application. This violates the Model-View-Controller paradigm we strive to keep when implementing GUI application such as this. All code relating to jobs or parts should be extracted to custom-made job and part classes and only the methods of these classes should be called from the UI code. This step is not required if the other improvements are made as they already address these issues as part of their implementation.

Conclusions

Considering what the project aimed to deliver compared with the end result I would personally claim the project a success. The required features the client requested were met and some additional features were also delivered. The main problem was that the client failed to stipulate in our interactions some important non-functional requirements, which would have resolved the issues we encountered once the solution was presented to the client. If I had been better aware of the need for a system that could continue operating in sub-optimal conditions then the client would have undoubtedly been happier with the solution. It would be unfair however to claim that this is solely the fault of the client. Had the project not suffered the set backs it did then the client should have seen a working version of the application long before the project deadline and the issues regarding data flow could have been avoided much earlier. I could also have been more pro-active in obtaining a full set of project requirements, which may also have helped resolve the issue.

Of most importance to myself is the fact the client was still happy with the solution presented and as a result the project should continue beyond the scope of a university project and become a commercial solution in itself.

As an application I am not particularly proud of the outcome. The implementation was rushed and although the solution works, it is sub-optimal. Given a little more time to polish the implementation and tidy some inelegant solutions I feel I could obtain a project with which to be proud of.

Reflection

This project has undoubtedly been a hugely beneficial process. I have added a new language to my repertoire (Objective-C), learnt the processes behind creating iOS applications and created my very first web service. These are all skills I would expect to be able to put to good use in the future. I feel however that these things are not the most important learning outcomes of this project.

Rather than individual skills themselves I have gained a new respect for the importance of the procedures we use. When attempting to implement a project of this scale it soon becomes unfeasible to "hit the keyboard running" as design issues soon become major problems if they have not been considered properly. I have also realised the importance of frequent communication if we are to be implementing a solution for a client. When writing code for oneself we are aware of exactly how we expect the system to be as we are the ones with the idea and the vision. When working for a client however it is very easy to misunderstand what is wanted or to miss important points that should guide the direction of the project. As a result priority should be placed on frequent and effective communication to reduce as much as possible an end result that does not meet the clients requirements or expectations. Making assumptions about how the project should work can easily result in solutions that do not meet the client's expectations and as a result one should always clarify any points that they are unsure of.

If I were to approach this project again I would make some important changes. Of highest priority would be to make sure the client provides as rich a set of requirements as possible (both functional and non-functional) and to clarify any points which they have not fully explained. I would also spend more time communicating with the client during the process such that any problems can be dealt with as soon as possible and in a manner that the client is happy with. I would also change the way in which I go about implementing features. Although taking shortcuts is often quicker at the time the problems these shortcuts produce are far greater compared to the small amount of time saved. I therefore feel that the features should be fully considered such that the most elegant solution is used from the outset. This reduces many problems that are encountered later in the project.

Glossary and Abbreviations

ATM (Automated Teller Machine): A computerized device providing automated financial transaction services to customers without the need to interact with a human cashier.

IDE (Integrated Development Environment)**:** An application that provides the user with all the necessary tools for developing software for a specific environment.

SDK (Software Development Kit)**:** A suite of software development tools which allow the user to develop software for a particular platform or operating system.

iOS (previously iPhone OS)**:** The operating system deployed on Apple iPhone and Apple iPad devices.

XCode: An IDE designed specifically for the development of iOS and OS X applications.

App Store: A service operated by Apple to distribute applications for iOS devices.

Throbber: An animated graphic to demonstrate that a background task is currently in progress. Unlike a progress bar it does not give an indication of how long the task may take or how much has been completed.

References

1. **Wenderlich, Ray.** How to Write A Simple PHP/MySQL Web Service for an iOS App. *www.raywenderlich.com.* [Online] 7 April 2011. [Cited: 12 March 2013.] http://www.raywenderlich.com/2941/how-to-write-a-simple-phpmysql-web-service-for-an-ios-app.

2. **Apple Inc.** Making HTTP and HTTPS Requests. *developer.apple.com.* [Online] 28 January 2013. [Cited: 23 May 2013.] http://developer.apple.com/library/ios/#documentation/NetworkingInternetWeb/Conceptual/NetworkingOverview/WorkingWithHTTPAndHTTPSRequests/WorkingWithHTTPAndHTTPSRequests.html.

3. **Wenderlich, Ray.** How to Write an iOS App That Uses a Web Service. *www.raywenderlich.com.* [Online] 12 April 2011. [Cited: 7 March 2013.] http://www.raywenderlich.com/2965/how-to-write-an-ios-app-that-uses-a-web-service.

4. **Copsey, Ben.** *allseeing-i.com.* [Online] 15 May 2011. [Cited: 28 May 2013.] http://allseeing-i.com/ASIHTTPRequest/.

5. **Brautaset, Stig.** SBJson for Objective-C. [Online] [Cited: 28 May 2013.] http://sbjson.org/.

6. **Bukovinski, Matej.** MBProgressHUD. *github.com.* [Online] 13 March 2013. [Cited: 28 May 2013.] https://github.com/jdg/MBProgressHUD.

7. **iPhone Dev Wiki.** *iphonedevwiki.net.* [Online] 11 December 2009. [Cited: 28 May 2013.] http://iphonedevwiki.net/index.php/UIProgressHUD.

8. *BCrypt Wikipedia Page.* [Online] [Cited: 1 June 2013.] http://en.wikipedia.org/wiki/Bcrypt.

9. **Fuerstenberg, Jay.** *http://www.jayfuerstenberg.com/.* [Online] 28 July 2011. [Cited: 1 June 2013.] http://www.jayfuerstenberg.com/blog/bcrypt-in-objective-c.

10. **ZBar.** ZBar Bar Code Reader. *ZBar Project Sourceforge page.* [Online] [Cited: 1 December 2010.] http://zbar.sourceforge.net/.

11. *curl Project Page.* [Online] [Cited: 1 June 2013.] http://curl.haxx.se/.