Emir Canpolat - ec363 - Dissertation

by E. Canpolat

Smartphone Application for Field Service Engineers for Canon Europe Ltd iOS Version

Author:
Emir Canpolat

Supervisor:
Dr. Michael Kampouridis

Words: 12000+

September 14, 2014

ABSTRACT

The aim of this project was to develop a mobile application that monitors the status of Canon printers. The application was intended to be a mobile alternative to the existing method in which field service engineers are able to view detailed information about printers, which is currently through a desktop website that is not optimised for mobile use. By following an iterative and incremental software development methodology, the resulting application is a fully functional system that exceeds the requirements set out during an initial meeting with software engineers and stakeholders at Canon UK headquarters. The application enables field service engineers to acquire real time data about printers at customer sites, including the number of events that have occurred on the device, current part counter information, and the firmware version of modules installed on the printer. The application also allows users to update the status of events that have occurred on the device, and post comments regarding the update.

The final version provides a substantial improvement to current way in which this functionality is accessed on the desktop website. This dissertation addresses the concerns of mobile application development in relation to the requirements of the current project, provides an in-depth walkthrough of the application's features and describes the significance of design choices made during development. Finally, the end product is analyzed and suggestions are made regarding future additions to the application.

CONTENTS

1. INTRODUCTION

Canon printers send real-time data to servers in Japan. This includes information such as the last communication date of printers, the state of consumable stock, such as papers and toners, as well as details of errors, jams and alarms that have occurred on the printer. Printers also send data regarding the parts installed on the device, including part replacement date, a counter value for its current usage and the recommended lifetime counter. There are also contact details of the device administrator at a customer site, in addition to the firmware version of modules installed on the device.

This data is incredibly useful for field service engineers, and as such, it is available for viewing on a website. However, this website is not user-friendly and is certainly not designed for mobile use, due to its poor navigation and layout, while being overcrowded with information (Appendix A). It is also prone to errors, in that it will log the user out of the system if a specified device ID does not successfully match with an actual device. There are also compatibility issues, as the website was developed for use on Internet Explorer 7. This issues render the website thoroughly unsuitable for mobile use at customer sites, and a more functional system is required for field service engineers to get a better overview of a device's status.

This project will attempt to create a fully functional mobile application that will provide users with an intuitive and user-friendly system on Apple's iOS platform. By improving access to this data and providing a core set of valuable features, it is expected that the application will increase productivity by taking greater advantage of the real-time data sent by printers. It will serve as an initial version that will allow Canon to continue developing the application independently. It is therefore vital that the author follows object oriented paradigms and that the code conforms to iOS mobile development coding conventions. It will provide users with a number of clearly defined features and serve as a means to get a quick overview of the status of a device.

The rest of this document will be organized as follows: Chapter 2 will present a literature review. Chapter 3 will detail the design of the application. Chapter 4 will provide an explanation of the implementation of the system's features, and finally, chapter 5 will offer an in-depth analysis of the final product.

2. LITERATURE REVIEW

2.1 Background

The use of mobile applications has been growing following the introduction of the iPhone in 2007 (Xu et al., 2011). Features including fast and reliable internet connectivity, multi-touch high resolution displays and powerful processors have allowed developers to implement highly functional mobile applications (Gavalas and Economou, 2011). This has boosted the number of mobile applications that bring content and services to consumers (Hammershoj et al., 2010), while it has also been beneficial for more specialized uses, including health services (Holzinger et al., 2011). For example, the Mobile Computing for Medical Graz system uses its custom-built graphical user interface to deliver an electronic solution to hand written patient records (Holzinger et al., 2011). The resulting high system usability proved to be very effective in terms of reengineering the workflows in the hospital, producing a 90% reduction in time spent by medical professionals and more importantly, a noticeable reduction in costs (Holzinger et al., 2011). Similar enhancements to the operations of a large multinational corporation such as Canon may provide similar reductions in time and cost.

Canon printers send real time information about their status to servers in Japan. Information such as the number of pages printed on the device and toner/ink levels are often used for the purpose of billing customers according to the usage data of these printers. However, these printers also send information that is particularly useful for Canon engineers at customer sites, allowing them to view up-to-date information about devices, such as the status of parts, consumable and events. The events generated by a Canon printer are as a result of errors, jams and alarms that have occurred on the device (the printer). These can include occurrences such as part failures, paper jams, part communication failure and low/empty stock of consumables such as paper and toner. Each event is accompanied by information regarding their handling status, which includes the current status, the handler's ID, the date and time of handling as well as comments from the status handler. A last communication date may also be useful in helping engineers prioritize the order in which printers are inspected. In this case, the engineer's first port of call would be to investigate why these printers are not sending data and

to restore communication.

Printers also send information that would allow engineers to pre-emptively determine whether parts need to be changed or when toners are close to running out. For example, if a part consumption value has come close to or exceeded its predefined recommended lifetime usage value, an alarm is generated and uploaded to the server, while this can also be viewed directly on a separate page. Additionally, information such as whether a part or toner/ink models are recommended by Canon or are a non-Canon brand may be useful in determining the quality and reliability of the part. Other information includes the firmware version of the main controller and of individual modules installed on the printer. These may allow the engineer to initiate a firmware update and determine whether certain events are inherently due to the software running on the printer.

This information, in addition to others, are valuable to field service engineers as it can can allow them quickly and accurately determine the status of printers at customer sites. For example, detailed error and alarm history provides insight into the lead up to existing errors (Nienaltowski et al., 2008), while data such as a printer part exceeding 80% of its recommended lifetime usage will provide engineers with an opportunity to check up on the part and carry out maintenance if necessary. In addition to errors generated by printers, API calls that are made to retrieve this data can also generate errors. These include errors such as failed login authentication, incorrect parameter lengths, non-existent company identification numbers in the database, and attempted access of data not shared with the logged in account (Error codes API).

The current website that engineers use to access this data is tailored towards traditional desktop use (Appendix A) and its user-interface is not optimized to meet the usage characteristics of modern mobile devices. The user-interface is recognized by Hong et al. (2002) as a crucial characteristic that affects a users' continued usage of a system. A poor user-interface was found to impair overall motivation to use a system, resulting in a direct influence on the amount of information extracted by the user (Crowther et al., 2004).

An effective user-interface is thought to provide different paths to individual features (Cho et al., 2009). For example, being able to view detailed device information should be available from multiple views of the application. A well-organized interface can enable users to quickly identify relevant information (Thong et al., 2006), resulting in users being able to use the system with greater ease (Cho et al., 2009). Similarly, A well designed user-interface is also an indication of good system

functionality, as it is the user-interface itself that enables the user to effectively achieve their goals by interacting with the underlying technology of the system (Cho et al., 2009). A highly capable system is often under-utilized if users are unable to intuitively access the functionality (Cho et al., 2009). In terms of judging whether a user is likely to continue using a system, studies by Igbaria and Tan (1997) and Bokhari (2005) show satisfaction to be a key factor, while a study by Cho et al. (2009) revealed that satisfaction is one factor that has a positive effect on continued usage, and it is well known that user satisfaction is governed by the experience of the user (Deng et al., 2010). Leger (2011) also highlights that the importance of user-experience due to usage rates of mobile applications, whereby one in four mobile applications, once downloaded, are never used again.

These studies indicate that there is a direct relationship between user experience, user satisfaction, and the continued usage of a system. User experience is one of these factors that the software developer has predominant control over and as such it will be the subject of focus throughout this literature review. The expected features of the application raises a number of challenges during development pertaining to user experience. The upcoming sections will initially begin with an investigation into the complexity of error messages in relation to how they can be presented intuitively. It will then continue with an investigation into the concerns relating to the implementation of the functionality outlined above, with a focus on maximizing the user experience.

2.2 Complexity of error messages

Concerning the complexity of error messages, Nienaltowski et al. (2008) investigated the level of detail in error messages in regards to the speed at which student programmers of varying experience levels, according to the Likert scale of 1 to 5 (Norman, 2010), interpreted the error messages. This study found that, irrespective of the student programmer's level of experience, a greater amount of information increases the likelihood that a programmer will be able to identify the error and suggest an appropriate correction (Nienaltowski et al., 2008). They also found a significant correlation between a higher degree of prior experience and the number of quick and correct answers (Nienaltowski et al., 2008). This was naturally attributed to the increased likelihood that such an error was previously encountered by the experienced programmer (Nienaltowski et al., 2008). Interestingly, no difference was found between the time it took to suggest an answer to an error when the presentation of the errors were shorter or longer, with the long form errors taking programmers an average of 24 seconds quicker to answer compared to questions provided in short form (Nienaltowski et al., 2008). This finding sug-

gests that although longer and therefore more descriptive errors may take longer to read, the additional information provides the user with greater depth into the error, resulting in quicker times to fully comprehend the error. The quickest answers though were from errors that also contained visual representations, with the average answering speed less than 35% compared to that of errors represented in long form (Nienaltowski et al., 2008).

The use of visually represented errors is also supported by Brown (1983), who indicates that error messages often do not portray the actual error in such a way that is both informative and direct. Brown (1983) suggests that this may be due to the contrasting amount of time spent on the quality of the error messages during software development, which is minimal compared to that spent on other functionality. Although software engineers often accept that error messages should be informative and assist with correcting the error, this is far from true in reality (Brown, 1983). As a solution to this issue, Brown (1983) recommends the use of a visual scale to represent the stage in which the error occurs. For example, an error with the paper feeding process may be portrayed as such by dividing the stages of a printing process into distinct phases and marking the paper feeding phase as the source of the error.

These results indicate that software engineers must find a compromise between highly detailed error messages that are often too long to read and therefore time consuming, and shorter error messages where the software engineers rely on the experience of the user/engineer to correctly identify the error and respond accordingly. A short message also lends its hand to mobile applications where screen space is limited (Kim et al., 2011). Based on this, combining short error messages with visual representations indicating where the error occurs in the printing process could be ideal for engineers to improve their efficiency when dealing with printer related issues at customer sites.

2.3 Data Visualization

As mentioned previously, Canon engineers currently login to a website that displays information about devices at a customer site. This website though does not provide a user-friendly interface when viewed on mobile devices. Data visualization is employed by software designers to communicate data to users of mobile devices in a more user-friendly manner (Isenberg et al., 2013). Despite being troublesome from a technical point of view, visualizations in mobile applications can increase the user's productivity (Isenberg et al., 2013). They can be helpful for the current project by being used to display information such as errors and consumption data

such as the current consumption value in relation to the recommended maximum usage value. Providing effective visualizations in mobile applications has its own challenges due to technical constraints such as limited screen size as well as the connectivity issues associated with the download of large amounts of data or data that is frequently updated server side (Chittaro, 2006).

A commonly cited issue by previous studies is avoiding graphics that look impressive but are difficult to understand and often mislead its users, while another is preventing users from getting lost while navigating on a small screen (Chittaro, 2006). There is also a similar issue that was previously encountered with the display of error messages; will an unnecessary amount of data increase the difficultly of comprehending the data, or will visualizing insufficient data leave the information up for interpretation (Chittaro, 2006). Visualization techniques that work well on desktop devices tend to fail on mobile devices due to limited screen space, difficulty of implementation and overall effectiveness (Chittaro, 2006), so it is up to the stakeholders whether they want their developers to dedicate time implementing visualizations.

The effective representation of text on mobile devices is particularly restricted due to the limited screen space (Chittaro, 2006). Dynamic text representation techniques that have been studied by Öquist and Goldstein (2003) include 'leading presentation' text that is horizontally scrolled on one line, and rapid serial visual presentation (RSVP), which is the successive presentation of chunks of words appearing at a fixed location. This study reveals that although leading text is initially more acceptable due to user familiarity, RSVP enables efficient reading of both short and long pieces of text (Öquist and Goldstein, 2003), despite the RSVP format placing greater cognitive demand on its user, with some words not being exposed long enough for cognitive processing (Castelhano and Muter, 2001).

Providing interactivity is also significant concern. Do you allow users to change what is displayed by adjusting parameters, and in that case, does the connectivity of the mobile device allow the application to seamlessly retrieve this information and update the view (Chittaro, 2006). You may also fetch this data in the background in anticipation that the user may eventually change parameters, but this would add unnecessary background processing, which may reduce responsiveness of the app (Yang et al., 2013).

2.4 Retrieval of large amounts of data

In addition to the issue presented by mobile interactivity, many of the application's potential features that were outlined in the background section, such as the viewing of detailed event information and firmware data, presents a similar challenge in regards to the retrieval of large amounts of data from the servers. This is due in part to the great amount of data that is returned with requests for this information. It is also partly due to the Canon data servers being located in Japan, so the large distance between Japan and Europe creates a noticeable delay in the time it takes for data to be sent and received (Clement and Stevenson, 2010). As a result, the response time is severely affected for users in Europe, which is a concern as the response time is critical in terms of maximizing the usability of the application. For example, a study by Forrester Consulting suggests that 40% of users will wait no longer than three seconds for a page to load before abandoning the application (Forrester, 2009). It is therefore necessary to consider the architecture of the application in regards to downloading this data while providing acceptable loading times.

While it is not feasible to duplicate servers in multiple locations to reduce latency, optimizing the application itself can help boost responsiveness for end users (Qian et al., 2011). For example, concurrent downloads are advocated by Kuschnig et al. (2011) as a reliable technique that can cut a proportion of the loading time associated with multiple requests for data. Another example is the Akamai edge server which parses and prefetches data to ensure that when the user requests the data, it is already in memory and can be loaded more quickly (Nygren et al., 2010). A similar method involves the pre-fetching of data that is embedded within links on the current page (Nygren et al., 2010). Although the two latter examples replicate responsiveness that users would expect with data that is hosted on local servers (Nygren et al., 2010), neither of these techniques would be suitable where data is frequently updated (Nygren et al., 2010), such as consumable printer parts or live printer event information. It can however, be useful for data that is less likely to change frequently (Nygren et al., 2010), such as company/customer/device identification (ID) numbers, and their associated detailed company/customer/device names. Another good way to handle this data may be to save the lists of customer, customer and device IDs and names to the device's storage, and loading them as and when they are needed.

Although the transfer of data can be optimised to handle large data, the connectivity of the mobile device is another consideration that the software developer must consider. Due to the differences between wireless networks and traditional desktop based environments running on ethernet connections (Wei et al., 2008),

mobile applications cannot make assumptions regarding the quality of the connectivity (Nicholson and Noble, 2008). Similarly, although the internet can be slow, it provides no guarantees regarding its reliability and performance (Nygren et al., 2010). Abowd (1999) lists some considerations that include whether differences in speed and security between telephone networks (3G, 4G) and local area wireless technology (Wi-Fi) affect the implementation of the application in terms of handling exceptions. Similarly, the higher likelihood of a dropped connection in cellular networks may require additional support that takes this into consideration (Abowd, 1999), while another consideration is assuring data integrity and its handling during potential loss of connectivity or power (Abowd, 1999).

Regarding the transmission of printer data to Canon's servers, a confidential document written by Canon details the timing and data size, in addition to a description of every possible transmission from printers to Canon's Central Server (Canon ImageWare 2012). It specifies that the frequency of regular transmissions from printers to the servers is once every 16 hours, with a maximum size of 250kb (Canon ImageWare 2012). However, when an event occurs, information about the event is transmitted when the event occurs, with a data size of up to 4kb (Canon ImageWare 2012). This means that as long as the printer is connected to the internet, data that is held by Canon servers is the latest. As a result, to allow the engineer to view the latest about printers at customer sites, the mobile application must also include functionality that supports the change in data. One issue is when to check for new data, whereby, considering the previously mentioned issue of latency, the intervals at which the servers are checked may significantly affect responsiveness. However, as the current API does not provide an option to check for changes, the only way to check for changes is to in fact request the data again, and to compare its values against the previous data. Once the data is retrieved, although it may make better sense to update the view regardless of whether it has been changed, this is not ideal as the engineer may not want the view to be updated. The best option would be to give the user complete control (Kangas and Kinnunen, 2005) as to whether the data is refreshed. This would involve adding a button to the screen that fetches and updates the data being displayed on the screen, regardless of whether the data is actually changed on the server. This would give greater flexibility to users as they would be able to refresh the data as and when they need to, while the number of connections made to the servers would be limited to when the user wants to refresh the data, thus reducing network usage.

Based on this research, it is evident that a mobile application with an optimised graphical user-interface and background processing may provide substantial ben-

efits for field-service engineers at Canon's customer sites. There is a distinct lack of existing, commercially available or otherwise, mobile applications that demonstrate such functionality, so there is. The remaining sections of this paper will focus on the design, implementation and analysis of a mobile application developed for Apple's mobile platforms.

3. DESIGN

This chapter details the design of the application. It begins by stating the requirements and describing the software engineering methodology that was employed during development. It then continues with a detailed overview of the applications's final features, with a number of screenshots of the application to help you visualize these features. The chapter ends with design considerations and UML diagrams that provide an insight into certain features of the application.

3.1 System Requirements

The initial requirements for this application were discussed and set out during a meeting with software engineers and stakeholders at Canon's UK headquarters. This meeting took place six weeks prior to the start of the application's development phase. Defining certain requirements at an early stage allowed the developer to fully understand the core requirements of the application, thus allowing adequate time for preliminary planning prior to the development stage. The planning involved research into iOS development and creating sample projects where features of the user interface were trialed.

Figure 3.1 presents a picture taken at the end of the meeting of a whiteboard displaying an initial storyboard of the application. The application starts at the login screen (3.1-A), and is followed by the device search[1] functionality (3.1-B) of manually searching through lists of companies, customers and devices, and continues with the detailed device information screen (3.1-C). This screen is intended to be the base for several features, including an events list (3.1-D), a detailed event info view (3.1-E), a parts status report (3.1-F), and a communication test (3.1-G). The communication test requirement was abandoned following an internal discussion at Canon Europe, due to a concern regarding the Universal Gateway (UGW) interface.

[1] This feature is located in the 'Browse' tab and not to be mixed with the device search feature found in the 'Home' tab, where users can directly specify a device to search.

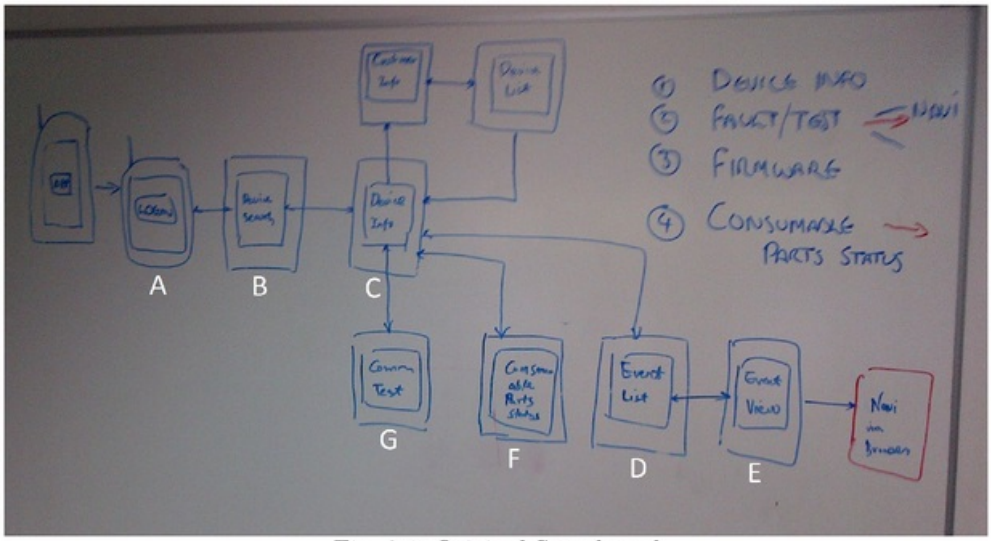


Fig. 3.1: Original Storyboard

The functional and non-functional system requirements are listed in table Appendix B. Certain requirements were set out at this meeting, while others were set during the development phase as a result of regular meetings and demos of the application to stakeholders, software engineers and field service engineers at Canon UK and Canon Europe. Other features, such as the inclusion of a tab view, were established independently by the developer as a result of design choices (an upcoming section) during development, while the functionality of the ensuing home tab was established by Canon.

3.2 Software Engineering Methodology

This application was developed by following an iterative and incremental software development methodology. This type of engineering methodology provides a great opportunity to modify the existing application during the development stage, as compared to other methodologies such as waterfall development (Larman and Basili, 2003). This was the preferred methodology as it paves the way for greater customization of the application towards the user's requirements, through their greater involvement in the development phase (Larman and Basili, 2003). However, at a higher level, the scrum methodology

Following an initial planning stage, development involved regular feedback, through teleconferences and meetings with project supervisors that led to incre-

mental additions and changes to the software. Feedback was gathered from a range of sources at Canon, including field service engineers themselves, as well as software engineers and project stakeholders. Providing these individuals regular access to incremental versions of the application led to the refinement of features and discussions regarding changes that would enhance the overall experience of the usability. Subsequently, this methodology improved the acceptance of the system as these individuals were provided with a significant amount of control in the development of the application.

Due to the author's inexperience in iOS development and objective-C programming, the iterative and incremental methodology provided an opportunity to develop stable versions of core features that were delivered early on. Following the investigation of enhanced iOS development conventions, these features were improved throughout the development phase, by implementing improved code efficiency and a designing a superior graphical user interface designs. For example, when static user-interface elements such labels were placed in table cells in the detailed device information view, the rest of the application's screens were eventually changed to adopt this style. As a result, existing features were continually improved during development, as and when an improvement to the current implementation was sought after, either by the developer or by Canon.

This methodology allowed prioritization of development time towards the most valuable requirements. For example, the first priority was the stability and usability of the application as a whole, to ensure that the system was intuitive and without fault. The priorities that followed this were the implementation of core features, changes to existing features, improvements to the source code, and finally, minor changes to the graphical user interface. This order or priority meant that existing features were in a fully working state before new features were implemented.

Critical path analysis is system that facilitates the planning of tasks that must be finalised before a development project can be declared complete. In this regard, the current project used its incremental methodology to add new tasks to its critical path, during the development of existing tasks or when completed tasks were reviewed by the team (the developer and Canon). These new tasks were often additions to previous tasks that improved the functionality of existing features. For example, in this project, although the base features of the event list and the subsequent detailed device information tasks were developed, a new task was added to the critical path of that of the events list, which was to allow users to change the time frame in which events were retrieved. Appendix I displays an overview of

the current project's critical path. The white coloured tasks demonstrate a walk through the application's features, which were often complete (with the expected functionality) at an earlier stage. In contrast, the yellow coloured tasks are tasks that were improved upon throughout the development of this application. As a result, the white coloured tasks were revisited, and completely rewritten in some occasions, multiple times during development to maintain the significance of the yellow coloured tasks.

A logbook was kept during the project, tracking the day to day progress during the development of the application. This comprised of all changes and additions that were made to the application, covering every day of the 3 month project duration. The logbook also comprises of a section describing functionality and significant changes made to the code during each version of the application. For example, in eMob v6, a pull to refresh feature was added to the screen that displays a list of error, jam and alarm events that have occurred on the printer. This was recorded as an application feature that was introduced in version 6. There is also a section that lists the major features that requested by Canon during development, such as the addition of an interface that allows users to change the handling status of an event. A similar section lists minor (and some not so minor) changes or additions that were either suggested by project supervisors at Canon following regular discussions, or that were discovered during development. These were things that could not be implemented during that day, so were recorded as a 'to-do' task that would be revisited at a later date. The day-to-day comments describe what was done during that day, as well as any problems that where encountered and anything that was related to the overall progress of the application. For example, while investigating how to support devices with larger screens, which was listed as a 'to-do' suggested by Canon following a teleconference, as noted in the logbook on the 26th of June, a minor issue with the user interface was fixed whereby the selection of the company ID list view resulted in an unsightly animation of the search bar moving up to take up the space of the search bar.

3.3 Application Functionality

When the application is first launched, the user is presented with a login screen (figure 3.2). At the top of this view is a segmented control that lets the user login to either the test or production server. Below these are text fields where the user can enter their username and password. In the event that that the user has forgotten their password, a 'forgotten password' button allows the user to contact Canon's support line either via email or phone (figure 3.3).

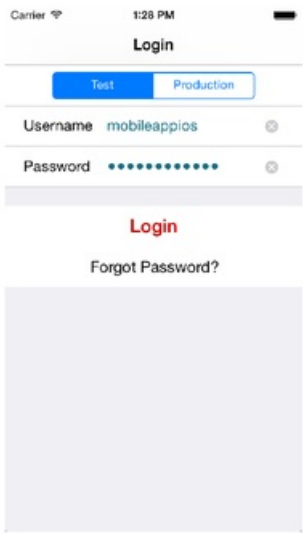


Fig. 3.2: Login View

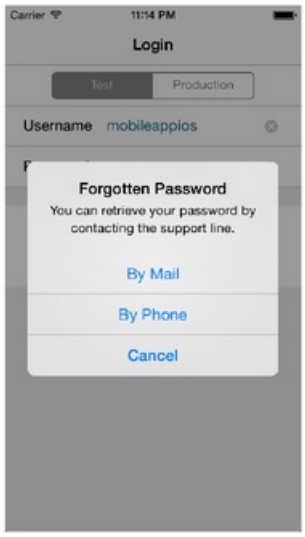


Fig. 3.3: Password View

The email option opens the iPhone's native Mail application (figure 3.4), with certain fields prefilled. Similarly, the phone option opens the device's phone application, but before doing so, the application displays the number that the phone will call and asks for confirmation (figure 3.5). Once the call has ended, the phone automatically returns to the application to provide a seamless transition. When the user attempts to login, and the application notices

Fig. 3.4: Email Application

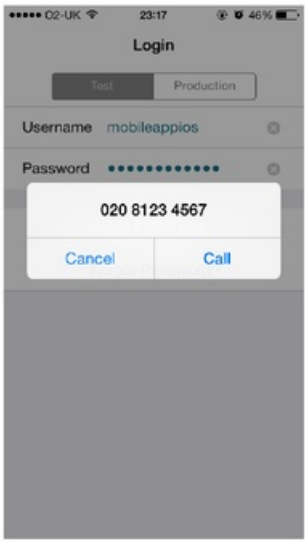


Fig. 3.5: Call Confirmation

that the user's password
has expired, this is handled by displaying a screen that allows the user to update their password (figure 3.6). Upon a successful login attempt, the user's username is saved to the device's storage and is loaded in a subsequent viewing of the login screen, as is the server that the user logged in to.

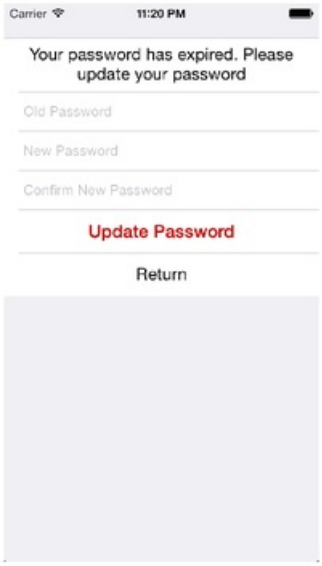


Fig. 3.6: Password Update

Fig. 3.7: Home Tab

Fig. 3.8: Password Expiring

Once logged in, the user is presented with a tab bar view that is pre-selected to display the home tab (figure 3.7). This tab serves as an introductory view that displays information including the user's name and the number of days remaining until the user's password expires. When this number is found to be less than or equal to 10, an alert informs the user and provides an option to update their password (figure 3.8), opening the same view to that of figure 3.6, but with a different title. The home tab also allows the user to search for a specific device by entering a device ID into a text field and initiating a search. Following a successful search, a detailed device information screen is displayed (more on this later).

Another way to reach the device information screen is by traversing a hierarchy of companies, customers, and devices. This feature is embedded within the Browse tab, which can be accessed by selected the Browse tab at the bottom of the tab bar view (figure 3.9). When selected, the application initiates a request to retrieve all companies that are available to the user.

The companies are then sorted in a depth-first hierarchy that depicts the master company of each company in the list. The sorted list starts at level 1(figure 3.9 A), which is the root company of the user[1], and continues with level 2, which is each company that has the root company as its master company (3.9 B). Each of these level 2 companies potentially has a set of level 3 companies that they are a master of (3.9 C). This continues recursively until all companies have been accounted for and placed in the hierarchy. The companies that are displayed in this list are filtered according to their company category - direct or indirect. This means that if the root company is direct, the company hierarchy is only populated with companies that are also direct. The same applies if the root company is indirect, where only indirect companies are listed. At each individual level of the hierarchy, the companies are also sorted alphabetically.

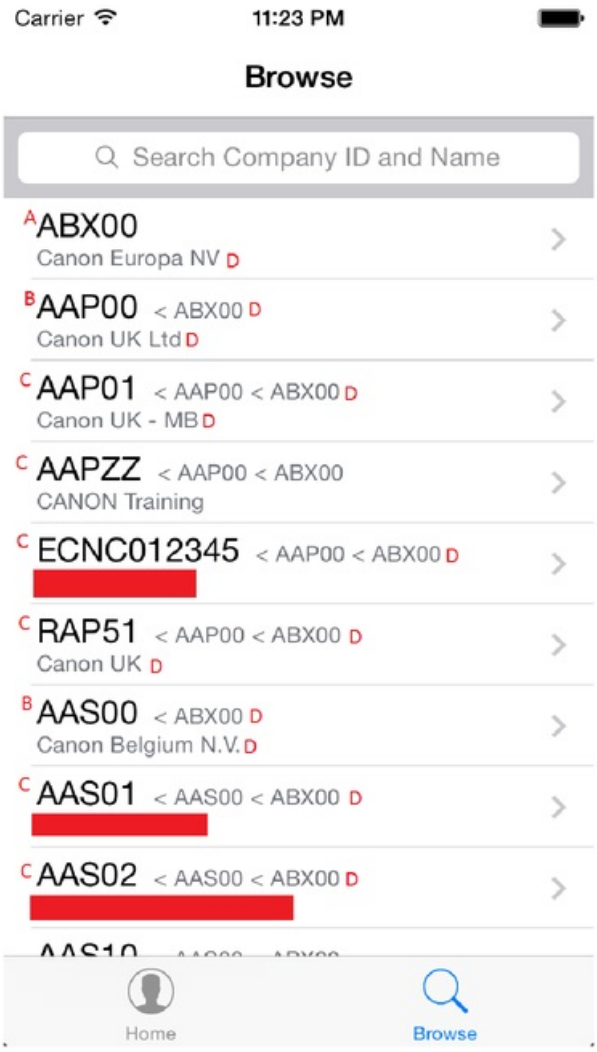


Fig. 3.9: Company ID List

[1] Here, level 1 is simply the root company available to the user. The actual level of the root company depends on the companies available to the individual user.

Fig. 3.10: Customer List

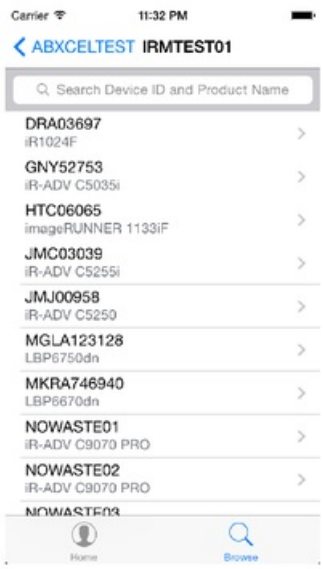


Fig. 3.11: Device List

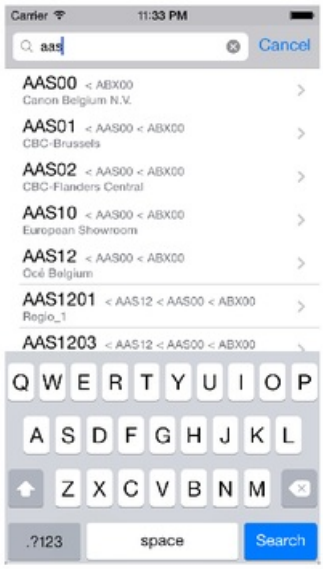


Fig. 3.12: Searching Lists

Each item in this company list presents the ID, name and the master companies of that company (figure 3.9 D) (Some companies are redacted here to protect confidentiality. These are selectable items that take the user to a list of customers that are attached to this particular company. This list (figure 3.10) is populated with the ID and name of each customer. These are also selectable items and take the user to a device list screen (figure 3.11) that displays the ID and product name of each device available to that specific customer and company combination. At each stage of the company, customer and device hierarchy, a search bar allows the user to filter each list according to the text that is entered. For the company and customer lists, this supports the ID and name, while the device list can be filtered by device ID and product name (figure 3.12).

Fig. 3.13: Detailed Device Information

By selecting a device ID, the user is presented with a screen that displays detailed device information about the selected device (figure 3.13), which includes the printer name, location,

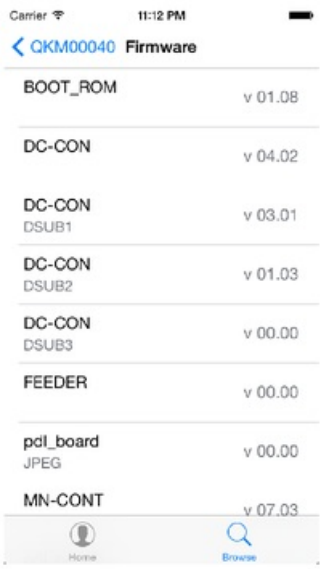


Fig. 3.14: Firmware List

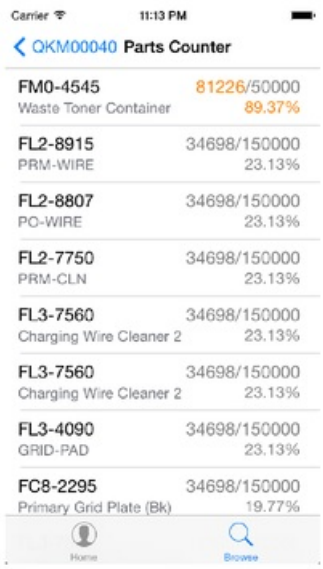


Fig. 3.15: Parts Counter

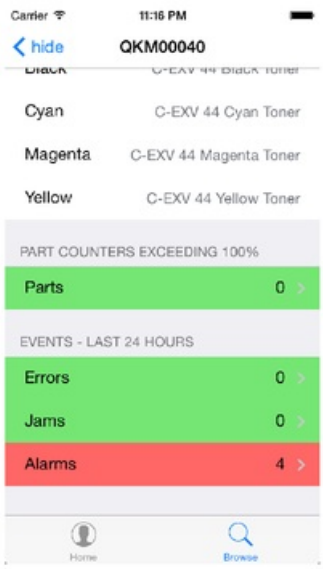


Fig. 3.16: Parts and Events

data about consumables and administrator contact details. These contact details can be selected to open the iPhone's phone or email applications. The email application is pre-filled with the administrator email address, while selecting a phone number asks for confirmation before directly calling the number. The device information view also displays the firmware version for the main controller of the device, and is a selectable item that opens a screen displaying a list of the firmware version of each module on the device (figure 3.14).

Another feature of the device information view is a consumable part report, where a count of the device's number of parts that exceed 100% of their recommended lifetime usage is displayed (figure 3.15). This is also a selectable item that opens a screen showing all parts installed on the device in descending order of consumption rate (figure 3.16). Here, the current lifetime counter and the consumption rate is colour coded to provide a visual representation of the status of each part. These colours match the colours that are currently used for the same feature found in the eMaintenance desktop website, providing the user with a familiar interface.

In the detailed device information screen, the application checks the number of the days that the device has not communicated with the servers. An alert is displayed if this number exceeds 3 days, and the background colour of the naviga-

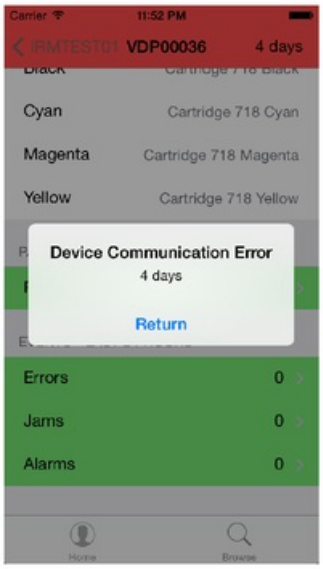


Fig. 3.17: Communication Check Alert

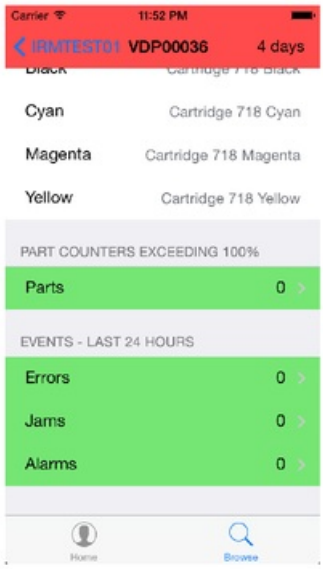


Fig. 3.18: Communication Check Aftermath

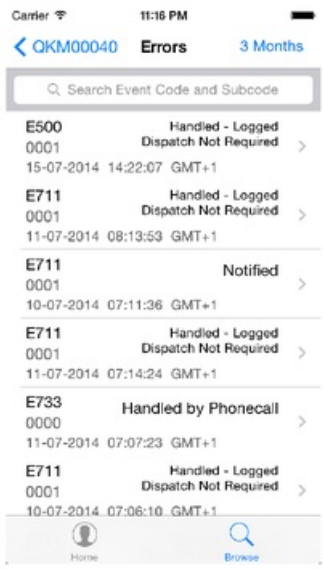


Fig. 3.19: Events List Screen

tion bar turns to a light red colour, while the right hand side of the navigation bar displays the number of the days that the device has not communicated with the server (figure 3.17). This is a permanent change (on this view) that is intended to capture the user's attention (figure 3.18). Due to the number of API requests that take place in this view, loadings times are reduced by an average of 40% with the introduction of simultaneous API requests.

The main feature of the device information screen is a section that indicates the number of errors, jams and alarms that have occurred on the device during the past 24 hours (figure 3.16). Selecting one of these event types takes the user to a screen that displays a list of events (of the selected event type) that occurred during the past 24 hours (figure 3.19)[1]. Here, the information includes the event code, sub-code, the occurrence date and its handling status. This screen also supports the pull to refresh UI gesture, which retrieves new events from the server and updates the list accordingly. By default, the application retrieves events from the past 24 hours.

[1] Here, the time frame has been changed to 3 months. Otherwise, the list would be empty in this case.

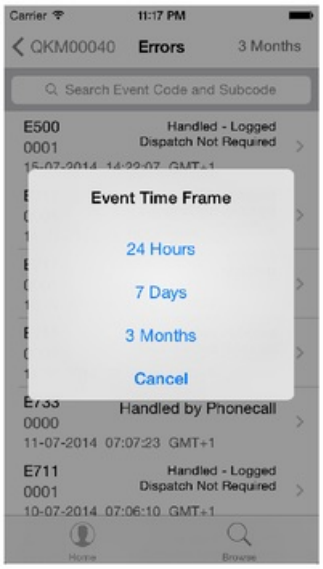


Fig. 3.20: Changing Event Time Frame

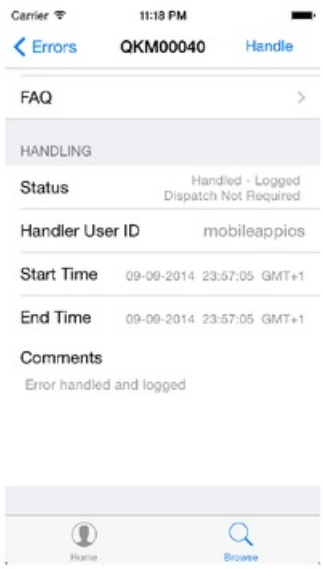


Fig. 3.21: Event Handling Status

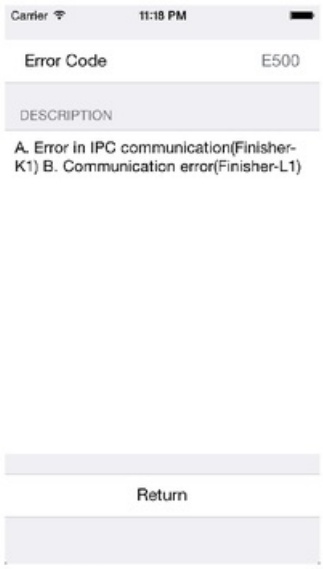


Fig. 3.22: Event Description Screen

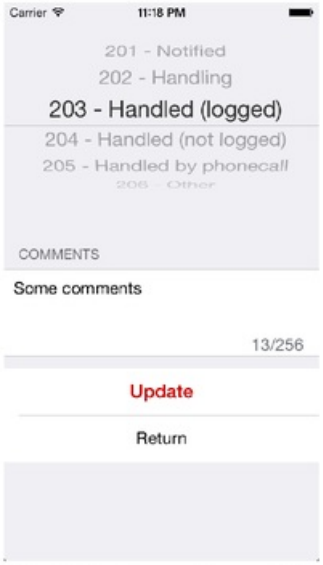


Fig. 3.23: Event Status Update

In the event list screen, the user is given an option to change the time frame from which the application retrieves events that have occurred on the device. These include the default 24 hours, 7 days and 3 months (figure 3.20). Once retrieved, the screen is updated to display all events that have occurred in the selected time frame. Each event is selectable, and takes the user to a detailed event information screen which displays additional information about the specific event (figure 3.26), detailed handling status information (figure 3.21) and a link to the description of the event in a seperate screen (figure 3.22). Figures 3.24, 3.25 and 3.26 show the difference between the event detail screens of errors, jams and alarms. Users can also change the handling status of an event by picking from a selection of statuses and filling an optional comments field (figure 3.23). To provide further information about each error event, a FAQ link is displayed in the error information screen that opens up a web browser that is preloaded to the FAQ page

of the error on the Canon eMaintenance desktop website.

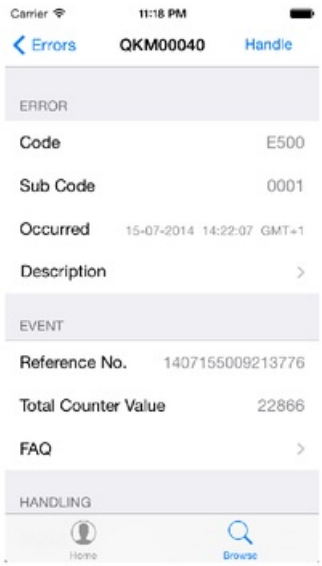


Fig. 3.24: Error Event

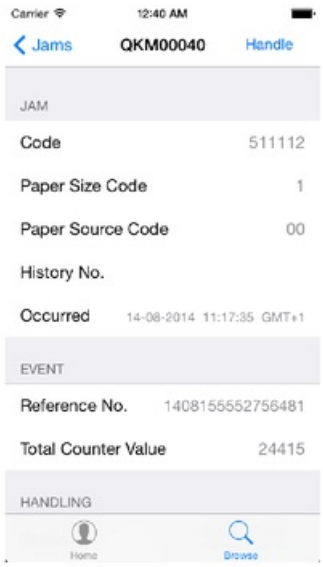


Fig. 3.25: Jam Event

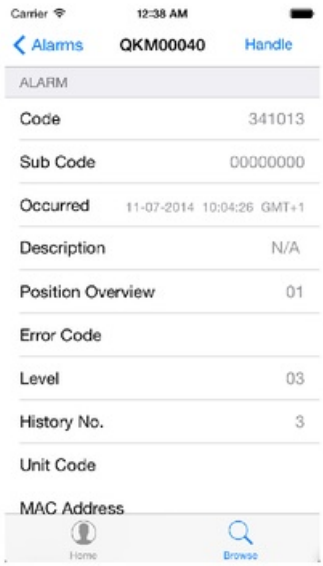


Fig. 3.26: Alarm Event

3.4 Design Considerations

There are many design considerations to bear in mind during the development of software. Addressing these considerations maximises the functionality of the application by developing software that is optimised towards the system's requirements. Some of the key considerations pertinent to this application are performance, usability, maintainability and extensibility. These considerations were addressed and are discussed in chapter 4, section 3.

3.5 UML diagrams

The following are diagrams created using the Unified Modelling Language (UML). These provide an insight into certain features of the final application. Figure 3.27 represents an activity diagram of the threading in the detailed device information screen. The threading begins by creating an NSOperationQueue, which is a built in API that provides control of concurrent operations. The queue is then limited to three concurrent operations, which is the number of API requests that the

Canon servers allow, for a single user at any one time. Six operations are then added to this queue. Once an operation is added, the queue beings the begins running its code, in the background, only if the number of operations that are currently running is no more than 3. When an operation has completed, the main thread, which is free to update the screen, handles the resulting data and updates the graphical user interface. This continues until every operation has run and completed. Once the developer provides the operations and its handling code, the queue is managed automatically.

Figure 3.28 is an activity diagram displaying the functionality of the login screen. First, the user is provided with options to select the server to login to and to enter their username and password. Next, if the login button is selected, the application runs a number of checks and then attempts to login if the checks are positive. If the login is successful, the application removes the login screen and displays the home tab. If it isn't successful, the application handles the error by extracting the error code and checking to identify if it is due to an expired account. If this is the case, the expired password view is displayed. Conversely the user is alerted with a description of the error, and allows the user to attempt another login.

Figure 3.29 is a state diagram of the implementation that checks for internet connectivity. Here, the application employs an API that continuously checks for connectivity and updates the value of a Boolean (true/false) to represent the connectivity status. This value is then read before every Canon API request, and handled appropriately.

Fig. 3.27: An activity diagram of the threading in the detailed device information view.

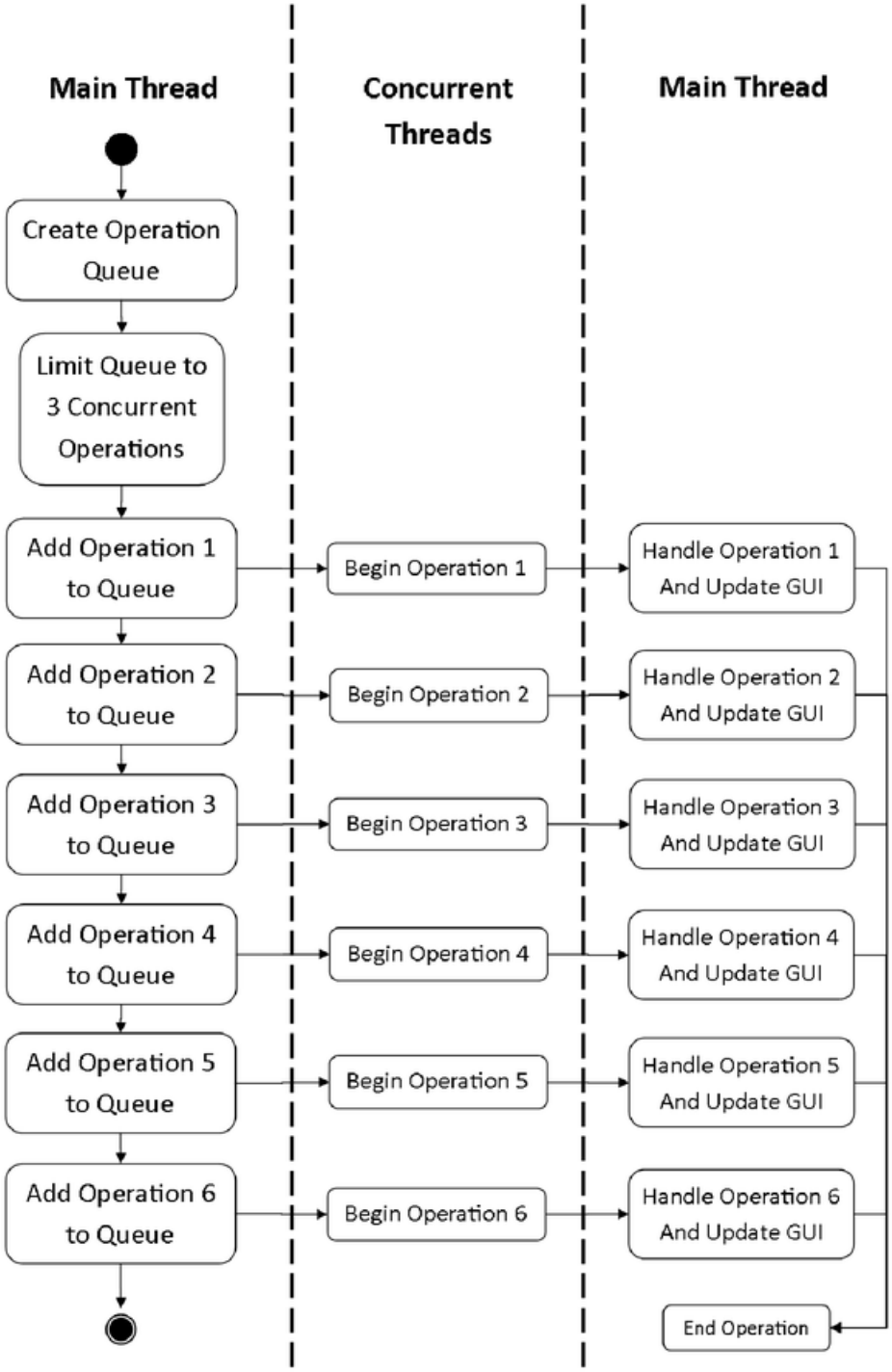


Fig. 3.28: An activity diagram of the functionality of the login view and error handling following an unsuccessful login attempt.

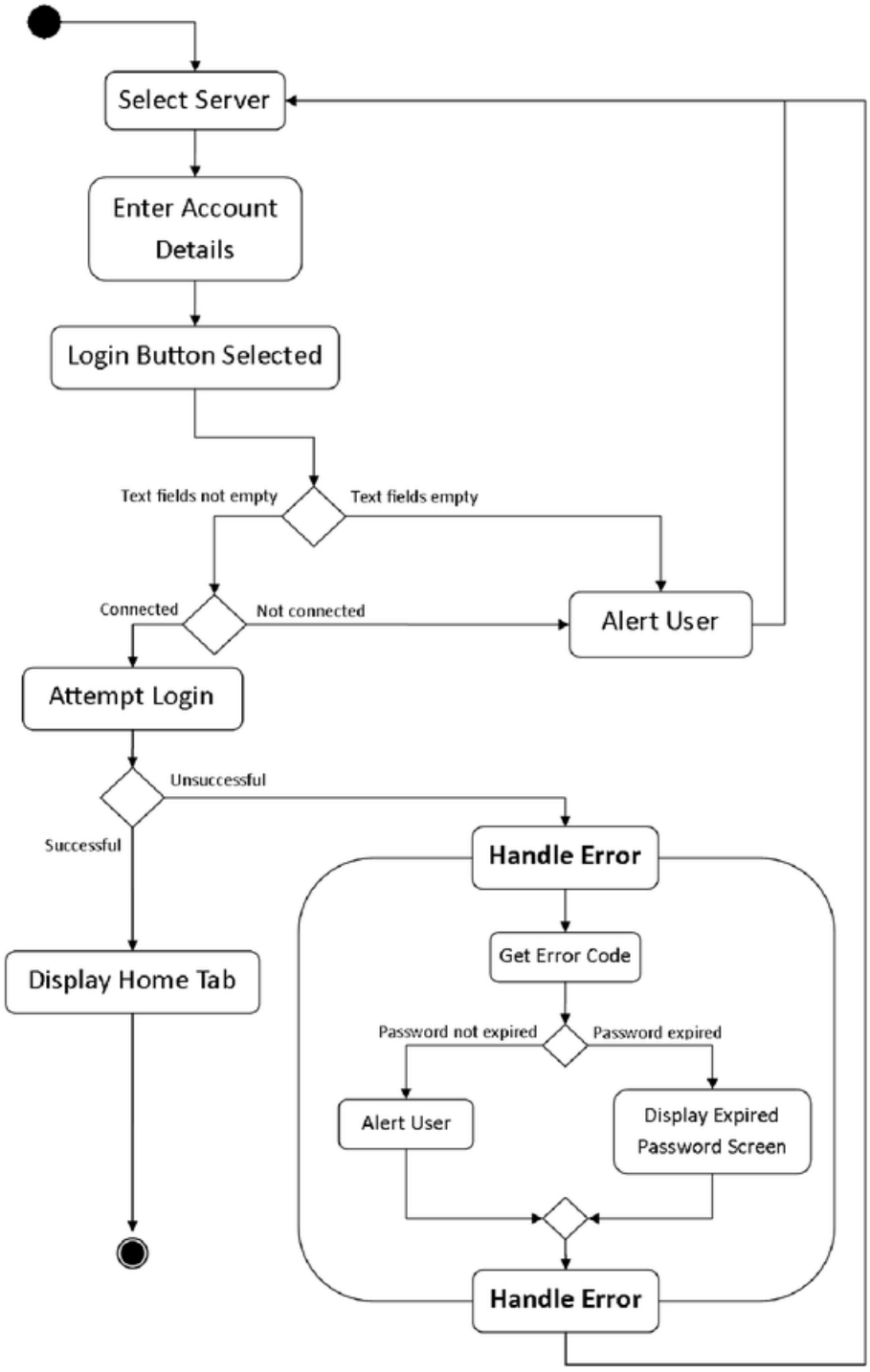
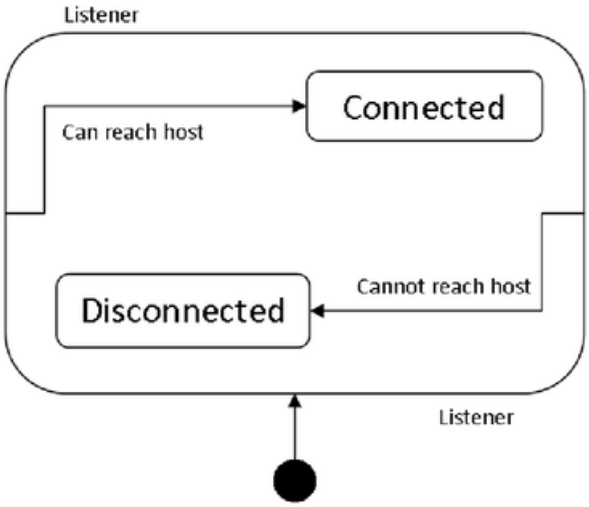


Fig. 3.29: A state Diagram of the internet connectivity implementation.



4. IMPLEMENTATION

4.1 Development environment

This application was developed using Xcode 5.1.1 on a MacBook 2013. The programming language that was used to develop the application was Objective-C. The new language, named Swift, was also a later option but was avoided as it was released only two days into development. The author had no previous experience in either Objective-C or mobile development and the source-code does not make use of any 3rd party software libraries.

It was agreed that an external revision control system would not be used due to the confidential API that the application utilizes. Instead, major versions (e.g. v1, v2, v3, etc.,) of the application were archived on a local machine and backed up on a redundant array of independent disks (RAID) configuration. These were accessed as and when needed to check the implementation of a previous working version.

4.2 Design Patterns

This application makes use of many design patterns that assisted the author in writing code that is easy to understand and reuse.

One of the most notable design patterns used in this application is the Model View Controller (MVC), whereby objects are categorized according to their role in the application. In this pattern, the model is an object that holds application data and defines its manipulation. The Event class (Corpus, section 3.7) is an example of a model object that handles the event data that is used by a number of controllers. Among others, the Event object holds the code, sub code and occurrence date of an event, the handling status code of an event and functionality that translates and returns the handling status code to a string.

The view is an object that defines the visual representation and the user interface to the model. In this application, the majority of views were created using

Apple's Storyboard feature (Corpus, section 8.11 and 8.12). This allows the developer to visually create views by dragging and dropping user interface elements such as buttons and labels, while also being able to create connections between views to visualize the appearance and flow of the entire application (if they are all created in the Storyboard). Each view is connected to an individual controller class, which is an object that accesses the model data and displays it in the view, while also containing the implementation for actions when the user interacts with the user interface. The EventListViewController (Corpus, section 2.9) is a view that displays the list of events that have occurred on the device by pulling them out of an array of Event objects (Corpus, section 3.7).

In the MVC model, the controller is notified by the model of any changes to the data, thus updating the view. The controller can also be notified by the view regarding any actions performed by the user, allowing the controller to update the model however necessary. For example, in the EventListViewController, when an event is selected, the view controller takes the Event object that was selected and transfers it to the controller of the subsequent view, and initiates a segue to the subsequent ErrorInfoViewController (Corpus section 2.6). This view then uses the Event object to fill the appropriate labels in its connected view. If the user happens to update the handling status of an event, in the UpdateEventStatusViewController (Corpus section 2.17), where the Event object is also transferred to, the Event object is updated and passed back, prompting both previous view controllers to update their respective views to display the Event with updated handling information. In the EventListViewController, the Event object in the array is also replaced with the updated Event object, to ensure that subsequent views maintain up-to-date event data.

4.3 Key Design Choices

Many design choices were made before and during the development stage that were intended to improve the functionality of the application. These design choices cover the usability, performance, maintainability and extensibility of the application.

4.3.1 Usability

To provide high usability, that is to make the application easy to use and intuitive, the application includes a number of small features. For example, the update handling status screen pre-selects the status picker to the current handling status of the event (figure 3.23), so as to indicate the current handling status of the event. Another feature of this screen is the character count at the lower right corner of the comments text field, which displays the number of characters currently entered into

the comment field against the character limit. The limit is enforced by preventing the user from entering more than 256 characters into the field, while the count turns red when the limit is reached. Another example of good usability is the ease in which the user is able to send an email or initiate a call with email addresses and phone numbers displayed in the application, such as the administrator contact details in the detailed device information screen (figure 3.13), and when contacting Canon support lines (figure 3.3).

The application also conforms to many iOS mobile conventions with the use of familiar user interface elements. For example, the events list screen supports a pull to refresh to update the list of events, while the application is based around a tab bar design. Both of these are common functionalities among the iPhone/'s default applications as well as popular applications such as Twitter and Facebook.

As this is an application that requires the user to connect to a server by logging in, there is a possibility that the session may have expired or disconnected. This can be due to a number of reasons, including the user changing their password while logged in, not interacting with the application, or by leaving the application as a background process for a prolonged amount of time. When this is the case, the API call returns an error code that indicates it as such. To handle this occurrence, in the event of an error the application checks for this specific error code, recognises the error code and notifies the user that this is case by displaying an alert box and immediately bringing up the login screen.

A confidential Canon document (The Mobile Handbook, Canon Europe 2014) outlines best practice and brand guidelines on how to approach mobile development for applications developed for/by Canon. The document highlights emerging trends from the present mobile landscape and provides iOS specific guidelines. At the design stage, the handbook states that it is important to lay out the complete functionality of the application with the use of a storyboard, working out how users will navigate through the application, and to involve the potential users in the design process to avoid technical flaws. Both of these were carried out for this project to ensure that the most valuable features are identified and built into the core application experience, while also planning ahead by ensuring that the application supports functionality in later versions.

The handbook also provides style guidelines. This includes the style of the splash screen and the positioning of the Canon logo on said screen, as well as the colour palette to be used throughout the application. This includes the hex colour of the red Canon logo, which is also used to create emphasis or to indicate an

interactive element, as well as the usage of white as the background, black for text throughout the application. It also states that mobile applications should be locked to portrait view in almost all screens, thus following the industry standard, and that it is important to avoid over-use of graphics and animations. These guidelines ensure that the application maintains brand consistency across Canon mobile applications. Icons and launch screen images were created by the author. These support the non-retina and retina 3.5 inch displays as well as the retina 4 inch display (figure 4.1). These conform to the guidelines provided by Canon and to the icon and launch screen dimensions set by Apple (Apple, 2014b). The application is also locked to the portrait view, and uses the red Canon colour to indicate an interactive element (figure 4.2).

Fig. 4.1: 4 inch Retina Launch Image

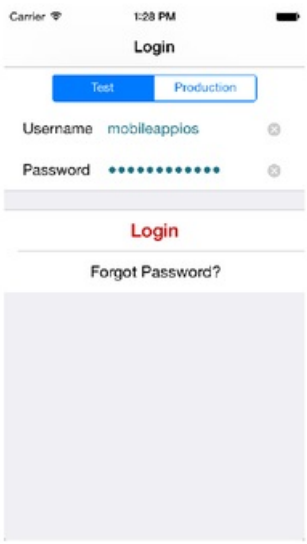


Fig. 4.2: Login Screen with Red Button

4.3.2 Performance

The performance of this application was improved by minimising memory consumption and loading times as much as possible. The latter is largely concerned with the API requests that are made to retrieve data from Canon servers in Japan. Partly due to the distance, there is often a noticeable delay when these requests are made, thus adding a disruption to many of the application's features. The software handles this by running these requests in the background, which keeps the main thread responsive to interactions with the user interface. While these requests are not an issue on screens where only one request is made, this is not the case on screens where more than one request is made. For example, the customer ID and device ID screens both make two API requests, which are run asynchronously as the second request uses data returned by the first. While there is little that can

be done to improve performance in these two screens, there is one screen where performance can and was improved.

The detailed device information view (Corpus section 2.5) makes six API requests, some of which take considerably longer than others. Originally, these were also run asynchronously, so there were significant loading times. As none of these requests are dependent on others, this was changed by running these requests concurrently, but were limited to three simultaneous requests, a limitation set by the servers. This was implemented by using Apple's NSOperationQueue API, which allows the developer to add operations to an operation queue, which are run concurrently in background threads. The operation queue was limited to three concurrent applications to conform to the server's limits, while the queue automatically starts the next operation when it detects that one of its operations have ended. Running the API requests in concurrent threads improved the view's loading speeds by an average of 55%, as represented by figure 4.3.

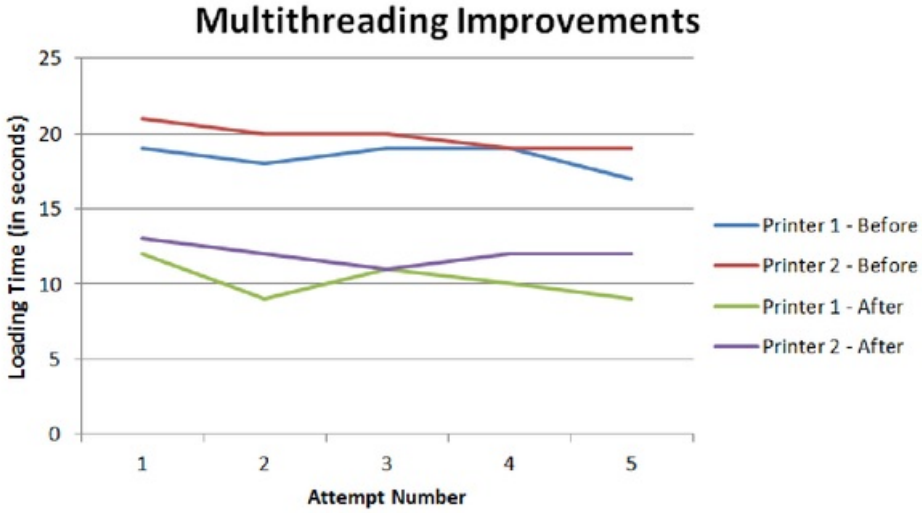


Fig. 4.3: Improvements in loading times in the Detailed Device Information Screen

This NSOperationQueue API was very useful as it improved upon the previous method in which the operations were run concurrently, where the threads were manually created and checked if they were completed. There were many issues with this, one of which was that the code was difficult to understand. This also contributed to another issue, whereby the extensibility (an upcoming section) was limited, as a lot of changes were needed if the developer wanted to add further

operations. By using the NSOperationQueue API, it is now far easier to add additional operations and to alter the maximum number of concurrent operations.

Another issue with the previous method was that there was no way to cancel the operations, once started, if the user navigated back from the detailed device information view. This meant that all six operations ran to completion even if the user happened to navigate back to the device ID list view when only three operations were started, and selected a different device. The NSOperationQueue supports the cancelling of operations and is achieved by simply invoking the built-in 'cancelAllOperations' function of the operation queue object within the view's 'viewWillDisappear' function, which gets called when the user navigates back from the view. As a result, the performance and efficiency of the system is improved as the software no longer runs surplus API requests.

4.3.3 Maintainability

Software maintainability is concerned with the ease in which the source-code can be maintained. This is due to a number of reasons, including the ability to meet new requirements, change the functionality of existing features, or to remove/replace components that are no longer required. These are important factors that also indicate the cost-effectiveness of adjusting the code or to rewrite it. This was achieved by utilising an object oriented design, with the use of MVC (described in chapter 4, section 2), and employing numerous utility/helper classes, which provide common functionality that is used throughout the application.

One example is a class that checks for an internet connection. When the application is first launched, the Internet class (Corpus section 3.10) checks the connectivity of the device by starting a notifier in the Reachability class (Corpus section 3.12). The Reachability class 'listens' to the connectivity of the device and changes the value of an anonymous enumerated type when the connectivity changes. This type is then used by the Internet class to set to value of Boolean indicating the connection status of the device. This value is then accessed before every API request to the Canon servers to confirm that the device is connected, so the application will only make the API request if the device is connected. When the device is not connected, the application informs the user that this is the case by displaying an alert box.

This implementation was largely taken from Apple/'s sample Reachability project, which is the recommended way to determine the connectivity of a device. This also improves the maintainability of the software as it is a common implementation. The Reachability class can also distinguish how the device is

connected to the internet, i.e. if device is connected to a Wi-Fi network or a cellular network such as 3G or 4G. Although the application currently does not change its behaviour according to the type of connectivity, this is something that can be used later on. This implementation prevents the need to repeatedly check for internet connectivity, thus having a limited effect on the application responsiveness. The application checks for connectivity by attempting to connect to Google. This is due to its high uptime, as it provided 99.984% availability in 2011, equating to an average of less than five minutes of downtime every month (Whittaker, 2011). After receiving feedback and discussing this with software engineers at Canon, it was agreed that it would be better idea to attempt a connection with the Canon server that the user is logged in to or is attempting to log in to. Despite the high connectivity rate of Google, it does not represent the availability of Canon/'s servers. However, when the selected server was used as the host, the Reachability class did not work as expected, so was replaced with Google.

Throughout the application, whenever API requests are made, there is a possibility that the parameters are incorrect or that the session is disconnected. In such a case, the response object returns an error code, which is an indication what the error is. The ErrorHandler class (Appendix C, Corpus section 3.6b) contains functionality that extracts and returns the error code in the form of an anonymous enumerated type, which are of constant relating to the type of the error (Corpus section 3.6a). For example, when the error code points to an expired password, the type is set to ExpiredPassword. When attempting to login and in the event of an error, the login view then checks for this type and displays the expired password screen if this is the case. The error handler class is also used to get a string representation of the error code. For example, the handler would return Your password has expired. Please update your password. This implementation makes the error handling functionality very easy to use and is easily expandable, by adding additional error types and their respective descriptions.

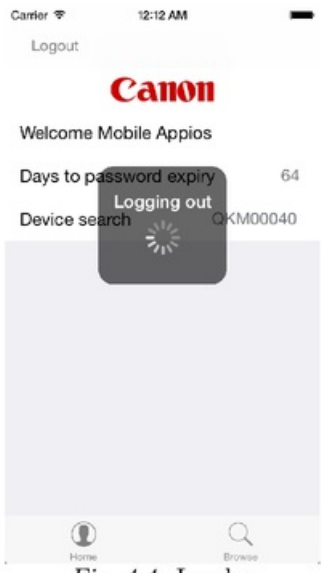


Fig. 4.4: Loader

Another implementation that improves maintainability is a class used to create large activity indicators when during certain loading events (figure 4.1). When the LargeActivityUIView class (Corpis section 3.11) is initiated (Appendix D), it creates a UIView object that holds a large activity indicator. To allow the

text to be changed, a setLabelText function can be called on the object, which sets the text displayed in the loader. This enables easy customisation as it can optionally be invoked before displaying the loader - however it does not need to be invoked, as the default is set to Loading. This implementation both minimises code duplication and makes it very easy to change the style of the loader.

Similarly, a separate class was created for the display of alerts (Corpus section 3.1). This class contains a static function that is called upon whenever an alert is displayed to the user. This function takes two strings as parameters, which is used to customize the text contained in the alert (Appendix F). The first is typically a short message in bold font that is placed in the top half of the alert, while the second is the text that is situated below this, where more detailed information can be displayed. Both messages do not have to be provided, as passing a null value to a parameter removes its section from the alert. This provides flexibility and ease of use to the developer as code is not replicated and the message can be easily customised.

Unlike the previous examples, some classes were created that maintain state and objects of these classes are passed between views. For example, a class (Corpus section 3.11) was created that performs various date and time related tasks. This includes the retrieval of the date and time 24 hours, 7 days and 3 months before the current date and time. It also holds the previous current date time that was retrieved, as well as a function that converts the date and time to a string, and one that calculates the difference in days between two dates. Some of these are used as parameters for certain API requests to retrieve lists of events that have occurred between the specified dates. Retrieving the last current date and time is used by the event list screen, where the pull to refresh feature allows the user to retrieve events that occurred between the previous current date and time and the current date and time.

4.3.4 Extensibility

Extensibility is concerned with the ability to add new functionality without making major changes to the architecture of the application. In this regard, the application originally featured a linear design, whereby following a successful login, the user was immediately presented with a view displaying a list of company IDs, and the subsequent views that followed this. Although this is where the majority of the application's functionality lies, it did not provide support for future additions to the application. While this wasn't a requirement that was requested by Canon, a replacement design was sought after as it was inevitable that features, not con-

sidered at this time, would be added in the future. Similarly, making this change early on in the development process would minimize any complications that would undoubtedly occur if the change was made to a larger application. To avoid disruptions to the progress of features desired by Canon, this was investigated and tested on a sample project before being implemented on the main application. This tab bar was something that was created by following sample projects before beginning to develop the Canon application, so the majority of the code was available and working, but needed testing and incorporating with other screens.

Approximately three weeks into development of the application, the user interface was completely revamped in favour of a tab bar design at the bottom of the view (figure 4.5). The tab bar became the base of the application and is displayed following a successful login attempt by the user, with the default view of the tab bar being the 'Home' tab. In addition to providing the ability to add separate tabs to the application, this change provided the opportunity to separate the different functions of the application. For example, originally the company ID list view contained a button that logged the user out of the application. Following the introduction of the tab bar, this was moved to the 'Home' tab, which is where the user would naturally expect to see a logout button. As pointed out in the logbook at the entry on June 25, implementing a tab bar as the base for the application led to a range of additional improvements to the source code. For example, public fields located in the tab bar controller were now accessible by all views that were pushed through the tab bar, which is approximately 90% of the application. This simplified many aspects of the code, one of which is the ability to call a function that displays the login view in the event that the session had expired.

Fig. 4.5: Example of the tab bar

To allow for internationalization, a resource file was created (Corpus section 9.1), which stores text displayed in the UI as a list of strings in a key:value format, such as "AccountLocked" = "Your account has been locked". To remove hard-coded non-UI strings, the majority of these are defined at the top of each implementation (.m) file, allowing the developer to adjust the value of strings with greater ease.

4.4 User-Interface

The majority of this application's user-interface was created using Xcode's Storyboard feature, where user interface elements, including views themselves can be dragged and dropped into place. Here, the developer is also able to create transitions between the screens.

Originally, the application presented static user-interface elements, such as labels and text fields in the login and detailed device information screens, on a plain white background (figure 4.6), much like elements on a website. However, this was not a user friendly design and was replaced early on in favour of table views, where user interface elements are embedded within static table cells (figure 4.7). In addition to being more attractive, they are also common among popular mobile applications such as Twitter, which gives the application a familiar design. They also provide greater flexibility by being able to change the order of elements very easily and are great for items that are intended to be selectable, as the entire cell is responsive to selection and a disclosure indicator can indicate it as such, while the selections can also be easily animated.

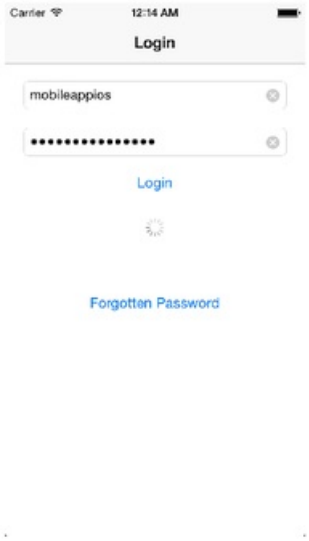


Fig. 4.6: Original User-Interface

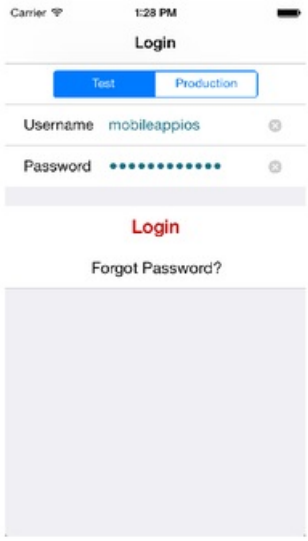


Fig. 4.7: Table Cell User-Interface

One of the core user interface elements of this application is a navigation bar. For the views that support this, which is the majority of the application, this is a bar that is permanently displayed at the top of the view, regardless of the scrolling position. Although its predominant purpose is to return to the previous view, through the use of a button on the left of the bar, it also adds functionality to certain views. For example, in the event list view it includes a button on the right that allows the user to change the time frame in which the application displays the events that have occurred on the device, updating the list accordingly and the label of the button to the selected time frame. Another example is the

device info view, whereby in the event of the device not having communicated with the server for more than 3 days, the navigation bar background turns red and the said number of days is displayed on a label on the navigation bar. The usually white navigation bar turning red is designed to attract the user's attention that the device has not communicated and to investigate and fix the issue.

The Apple developer documentation provides Alert Views as a concise and informative way to convey important information (Apple, 2014a). This alert view also allows the user to make a decision about a course of action (figure 3.20). For example, in the login screen, if the user forgets their password, a 'forgotten password' button invokes an alert view where the user is provided with options as to how to contact Canon's support line.

4.5 Testing

Implementation of this application involved a considerable amount of black-box 'manual' testing. It was concerned with ensuring the usability of the application, to check if the user interface is easy to use and understand, while making sure that the application. During this testing, many discoveries were made that allowed the author to find and fix errors in the code that led to unexpected results or to a system crash. For example, the feature that allows users to search for a specific device by specifying a device ID would, in most cases, would produce an expected result by displaying a detailed device info view of that device. In other cases, the detailed device information view would be of a different device. After an investigation, the issue was isolated to the function that handles the data returned by the API call, whereby the code was assuming that the desired device info data would always be present at the first object of the returned object of device information. Due to an inconsistency of the Canon database, this was not an issue when the parameters of the API call were able to match a specific device. While in other cases, a device would not be matched so all devices of a customer were returned, with the first object not necessarily being the desired device. This was then easily resolved by checking that the device ID in an object of the returned array matched the specified device ID, and using that object to populate the detailed device information view.

A limited number of uni tests were also created, testing certain functionality to determine if they are fit for use (Corpus - Section 6). The application is to be tested by a professional software Q&A team at Canon prior to its full release. Prior to that, the application is going to be given to two Canon resellers for feedback and testing of the system at real customer sites. As a result, significant white box

testing procedures were not a requirement of the development process, although unit testing was carried out.

5. ANALYSIS

This chapter will address the positive and negative aspects of the implantation of this project in relation to the key requirements of the application, which include usability, stability, maintainability and extensibility. In addition, this chapter will address the limitations of the final product and provides suggestions for future development of the application.

5.1 End product

The final version of the application provides a substantial improvement to the previous fashion in which field-service engineers were able to view printer data. This is achieved by providing a user interface that is developed around a set of clearly defined features, as discussed in the literature review. Although the features detailed in the original storyboard (figure 3.1) remain in the final product, the storyboard was substantially refined and expanded upon during development. The final storyboard (Appendix F) contains several more screens and greater interactions between these screens. The software also takes account the numerous situations in which faults can occur, all of which are handled appropriately. For example, in the event of an error, the application always checks for a session disconnection error code, and handles this occurrence by resetting the application and displaying the login screen. The system also supports an extensible design that allows the future developer to seamlessly make changes to the existing functionality, or to add new features to the application.

There are a few instances of findings resulting from the literature review not being employed in the final version. For example, the literature review suggests that an error is interpreted more quickly by combining short error messages with visual representations, such as the use of a visual scale to represent the stage in which an error occurs. This was not attempted by the developer of this application due to the large number of errors, jams and alarms to account for. This would have been potentially very time-consuming which would leave the developer with less time to implement and optimize other features, in addition to being a feature requested by Canon. Another matter in the literature review that proved to be

unused in the final application is the dynamic representation of text. This was not required as the developer devoted greater effort to designing the views around the text that is displayed, often by entirely rewriting the views, rather than adapting the text to fit within the view.

5.2 What went well

To maximize the usability, stability, performance and maintainability of this system, approximately one third of the development time was spent entirely on improving upon the application's existing features. This meant that almost 95

This was a by-product of the iterative and incremental software development methodology that was employed, as it leans towards working features before the quality of the code (Larman and Basili, 2003). This was intended to allow the stakeholders at Canon to view initial iterations of their requirements, thus maximising their opportunity to make suggestions or change requirements before substantial time was spent on enhancing features that might have been scrapped. Once the features and requirements were finalised towards end of the second month of development, the subsequent period comprised of significant changes to the source-code and user interface. One of the major changes was a complete revamp of the application's multithreading implementation. As discussed previously in chapter 4.3.2, the largest such change was made to the detailed device information view, where loading times were improved by up to 55%. Other significant changes were targeted at the implementation of the internet connectivity check and for the error handling of API requests (Chapter 4.3.3).

The key aim of this period was to conform to iOS development guidelines. For example, this included a stricter implementation of the Model-View-Controller design pattern, and an expansion of delegate functions, which are used to pass data back to the view controller of the preceding screen. A direct product of these two changes was the vastly improved implementation of the feature that updates the information in previous views when the handling status of an event is updated by the user. This was achieved by creating an object (model) of each event, containing information about the event, and passing these between the three view controllers that used the event data information. More on this in (maintainability section in design pattern). The source code was also heavily refactored to follow object oriented programming paradigms such as loose coupling, avoidance of code duplication and the use of recursion. An example of the recursion can be found in the company ID list view (Appendix G, Corpus Section 2.2 starting at lines 217 and 274).

Two weeks prior to the end of the development phase, the source-code was reviewed by the lead software engineer at Canon Europe, who provided suggestions regarding improvements pertaining to common Objective-C and iOS mobile development conventions. This feedback was extremely valuable due to the authors inexperience in Objective-C and iOS mobile development. Some of the suggestions included the use of non-hard coded strings, use of a builtin folder that manages all images throughout the application, correct ordering of property modifiers, use of forward class declarations as well as the use of '#pragma mark' to mark up protocol functions. It also included the expansion of the internationalisation support by including more of the text displayed in the user interface. As a result of these suggestions and subsequent improvements, the final application is a far more complete system.

5.3 Limitations

A limitation of this application is the lack of functionality when the mobile device is not connected to the internet. Although this is to be expected from an application that retrieves and displays data stored on remote servers hosted on the internet, no internet connectivity effectively renders the application inoperable. There may cases, however rare, that the user's device cannot establish an internet connection, while there are also numerous reasons for poor internet connectivity. This issue is present with many existing popular applications, whereby an offline mode is not provided, and a loss in connectivity is handled by locking the user out of the system until connectivity is established. In the current application, internet connectivity is checked before each request and a negative result is handled by displaying an alert. Despite this limitation, this application does endeavour to provide as much offline functionality as possible without being an intended feature. For example, users are able to pre-load any screen, likely a detailed device information view, while connected, and the page will still be available when not connected as long as users don't navigate back.

5.4 Remaining Issues

An issue present with the current application is the temporary display of the tab bar view when it is initially launched. This is due to the tab bar view being the application's default view, with the login view being displayed by animating over it. This is implemented so that the login view is immediately displayed when the application is launched. However, there is sometimes a short delay, particularly with older devices, causing the tab bar view to be displayed for a very short period

of time. The reasoning behind this implementation was that, in the event that the user's session times out, rather than resetting the application and displaying the login view, the application would maintain the previous state of the session if the same user logged in. However, this was not implemented due to time constraints.

While this is not a significant issue and there is no security issue resulting from this, it is something that should be resolved before the application is fully released. A short-term fix would be to cover the tab bar view with an empty white view for the first launch of the application until the user has successfully logged in. A longer term fix would be to redesign the application's architecture so that the login view is set to the default view and to display the tab bar view only following the first successful login attempt. Although this is a fairly straight forward fix that would maintain the feature described above, time constraints reduced the priority of this fix.

Another issue is the loading times of the application during an API request, taking considerably longer on an iPhone 4, compared to the iPhone simulator on the MacBook Pro. The iPhone 4 loads at a speed that is often 5 as long as the simulator. Although this may be due to the age of the device, there is a possibility that the loading times are similar on newer version of the iPhone, just not to such an extent.

5.5 Encountered Difficulties

During the development of the application, there were a few issues that caused minor delays to the progress of the application. Early on, the server that the API was connecting to needed to be changed by setting the URL in the code. This was an issue that presented itself on the first of development and was addressed the next day. There were also issues with the API that was generated, in that certain calls to request data returned null values (Logbook - 12 June). The team at Canon managed to fix this in 4 days. The issue was identified on a Thursday, reported during a meeting the following day and fixed on Monday. Due to the weekend, Canon only started investigating this on the Monday and provided a solution on the same day. However, as the developer of this application continued working did not stop development on weekends, in order to prevent further delays, the developer continued working on other features during the weekend, such as list filtering.

Following the fix, as these errors in the API code could only be discovered when they were first used, there was a small delay between finding an error and

determining that it was due to the API code. However, over time the API code issues were easily recognisable so were no longer an issue. Each change to the API was noted down to ensure that any changes to the API, following an update to the Canon systems for example, the regenerated API code could be fixed again with these noted changes. Another issue was being unable to login to the servers due to server downtime over a 3 day period. Due to the nature of the application, significant changes could not be made to the majority of the application without having access to data from the servers. This was a good opportunity to make enhancements to the code. However, this was also difficult as any change would need to be tested to ensure it was functional and that it did not break other functionality. Without the data this was very time consuming and was mostly avoided.

Other difficulties were mostly those that are to be expected during the development of software, especially by a developer that is inexperienced in mobile development and in the Objective-C programming language. When other matters, such as the clarification of certain features or following the encounter of a feature that looked to have its implementation changed, the project supervisors at Canon were contacted for a swift resolution. To avoid further delays, the developer of this application continued working on other features until responses was received. All of these issues and minor delays are chronicled in the daily notes of the project logbook. It is important to note that none of these difficulties caused significant delays to the development of this application.

6. CONCLUSION AND FUTURE WORK

The aim of this project was to develop an initial version of a mobile alternative to the Canon eMaintenance website, which was successfully achieved. The final application provides an extremely valuable set of intuitive and stable core features, as well as a design that supports future development, which was realized by making every effort to achieve a high degree of maintainability and extensibility. There are significant opportunities for future work to add functionality to the application and to address the concerns raised in the previous chapter. The application will undergo testing at Canon and will eventually be released to field service engineers.

One of the features that would enhance this application is the introduction of offline functionality. This could take many forms, but one example that was discussed during a meeting is the option to update the status of events from a non-connected location. When the user selects from the list of statuses, and optionally adds comments, upon pressing the update button, rather of blocking the update, the user would be presented with the option to save the details of the update to the device. When network connectivity is restored, the user would be presented with the option to send the saved update to the server. Another offline feature would allow users to save device information screens to a third tab on the application. This tab would hold all saved devices in a list that displays the device ID and the date of when they were saved. These device information views could be pre-loaded and saved to this list, and selected when offline to view the device. This type of offline functionality would extend the application's usability (reference 99 on Dis firefox folder) and limit the downtime of engineers by continuing to update event statuses and viewing detailed device information.

The previous feature could be expanded upon by converting it into a 'favourites' tab. This would allow users to add devices that they regularly visit through a button on the detailed device information view, thus negating the unnecessary loading times associated with loading company and customer IDs. If the device is connected to the internet, upon clicking a device in the favourites tab, the application would retrieve and display the latest data about the device, while conversely, the previous device data would be loaded. Interactivity would also allow users to update selected device info views or update the entire list. This may be expanded

to save favourite companies and customers, with both having separate views in the favourites tab. This functionality would provide great improvements to the application's usability by providing a more personal and customized experience.

It may also be a good idea to save the list of companies, customers and devices to the system's storage as they are highly unlikely to change. These lists would then be displayed directly. They could still be downloaded when the respective screens are opened and by displaying a loader to illustrate this, updating the view when it is retrieved.

Another feature that would improve the application would be to display the last communication date of each device (printer) in the device ID list. The background of the cells would then be coloured red when the device's last communication date exceeded 3 days and a label could be added that displays this number. The last communication date is a significant number that an engineer uses to prioritise which printers to inspect first. Displaying this number before selecting a specific device in the list would allow engineers to identify these devices in a timelier manner. This was a feature that was planned individually by the developer but was not implemented due to time constraints.

Bibliography

Abowd, G. D. (1999). Software engineering issues for ubiquitous computing. In *Software Engineering, 1999. Proceedings of the 1999 International Conference on*, pages 75–84. IEEE.

Apple (2014a). Alert Views. https://developer.apple.com/library/ios/documentation/UserExperienc

Apple (2014b). iOS Human Interface Guidelines - Icon and Image Sizes. https://developer.apple.com/library/ios/documentation/UserExperience/ Conceptual/MobileHIG/IconMatrix.html.

Bokhari, R. H. (2005). The relationship between system usage and user satisfaction: a meta-analysis. *Journal of Enterprise Information Management*, 18(2):211–234.

Brown, P. J. (1983). Error messages: the neglected area of the man/machine interface. *Communications of the ACM*, 26(4):246–249.

Castelhano, M. S. and Muter, P. (2001). Optimizing the reading of electronic text using rapid serial visual presentation. *Behaviour & Information Technology*, 20(4):237–247.

Chittaro, L. (2006). Visualizing information on mobile devices. *Computer*, 39(3):40–45.

Cho, V., Cheng, T., and Lai, W. (2009). The role of perceived user-interface design in continued usage intention of self-paced e-learning tools. *Computers & Education*, 53(2):216–227.

Clement, A. and Stevenson, J. H. (2010). Regulatory lessons for internet traffic management from. *Global Media Journal – Canadian Edition*, 3(1):9–29.

Crowther, M. S., Keller, C. C., and Waddoups, G. L. (2004). Improving the quality and effectiveness of computer-mediated instruction through usability evaluations. *British Journal of Educational Technology*, 35(3):289–303.

Deng, L., Turner, D. E., Gehling, R., and Prince, B. (2010). User experience, satisfaction, and continual usage intention of it. *European Journal of Information Systems*, 19(1):60–75.

Forrester (2009). ecommerce web site performance today - an updated look at consumer reaction to a poor online shopping experience. commissioned study.

Gavalas, D. and Economou, D. (2011). Development platforms for mobile applications: Status and trends. *Software, IEEE*, 28(1):77–86.

Hammershoj, A., Sapuppo, A., and Tadayoni, R. (2010). Challenges for mobile application development. In *Intelligence in Next Generation Networks (ICIN), 2010 14th International Conference on*, pages 1–8. IEEE.

Holzinger, A., Kosec, P., Schwantzer, G., Debevc, M., Hofmann-Wellenhof, R., and Frühauf, J. (2011). Design and development of a mobile computer application to reengineer workflows in the hospital and the methodology to evaluate its effectiveness. *Journal of biomedical informatics*, 44(6):968–977.

Hong, W., Thong, J. Y., Wong, W.-M., and Tam, K. Y. (2002). Determinants of user acceptance of digital libraries: an empirical examination of individual differences and system characteristics. *J. of Management Information Systems*, 18(3):97–124.

Igbaria, M. and Tan, M. (1997). The consequences of information technology acceptance on subsequent individual performance. *Information & management*, 32(3):113–121.

Isenberg, P., Isenberg, T., Hesselmann, T., Lee, B., Von Zadow, U., Tang, A., et al. (2013). Data visualization on interactive surfaces: A research agenda. *IEEE computer graphics and applications*, 33(2):16–24.

Kangas, E. and Kinnunen, T. (2005). Applying user-centered design to mobile application development. *Communications of the ACM*, 48(7):55–59.

Kim, K. J., Sundar, S. S., and Park, E. (2011). The effects of screen-size and communication modality on psychology of mobile device users. In *CHI'11 Extended Abstracts on Human Factors in Computing Systems*, pages 1207–1212. ACM.

Kuschnig, R., Kofler, I., and Hellwagner, H. (2011). Evaluation of http-based request-response streams for internet video streaming. In *Proceedings of the second annual ACM conference on Multimedia systems*, pages 245–256. ACM.

Larman, C. and Basili, V. R. (2003). Iterative and incremental development: A brief history. *Computer*, 36(6):47–56.

Leger, B. (2011). First Impressions Matter! 26% of Apps Downloaded in 2010 Were Used Just Once. www.businessinsider.com/chart-of-the-day-how-many-users-does-twitter-really-have-2011-31/3.

Nicholson, A. J. and Noble, B. D. (2008). Breadcrumbs: forecasting mobile connectivity. In *Proceedings of the 14th ACM international conference on Mobile computing and networking*, pages 46–57. ACM.

Nienaltowski, M.-H., Pedroni, M., and Meyer, B. (2008). Compiler error messages: what can help novices? In *ACM SIGCSE Bulletin*, volume 40, pages 168–172. ACM.

Norman, G. (2010). Likert scales, levels of measurement and the laws of statistics. *Advances in health sciences education*, 15(5):625–632.

Nygren, E., Sitaraman, R. K., and Sun, J. (2010). The akamai network: a platform for high-performance internet applications. *ACM SIGOPS Operating Systems Review*, 44(3):2–19.

Öquist, G. and Goldstein, M. (2003). Towards an improved readability on mobile devices: evaluating adaptive rapid serial visual presentation. *Interacting with Computers*, 15(4):539–558.

Qian, F., Wang, Z., Gerber, A., Mao, Z., Sen, S., and Spatscheck, O. (2011). Profiling resource usage for mobile applications: a cross-layer approach. In *Proceedings of the 9th international conference on Mobile systems, applications, and services*, pages 321–334. ACM.

Thong, J. Y., Hong, S.-J., and Tam, K. Y. (2006). The effects of post-adoption beliefs on the expectation-confirmation model for information technology continuance. *International Journal of Human-Computer Studies*, 64(9):799–810.

Wei, W., Wang, B., Zhang, C., Kurose, J., and Towsley, D. (2008). Classification of access network types: Ethernet, wireless lan, adsl, cable modem or dialup? *Computer Networks*, 52(17):3205–3217.

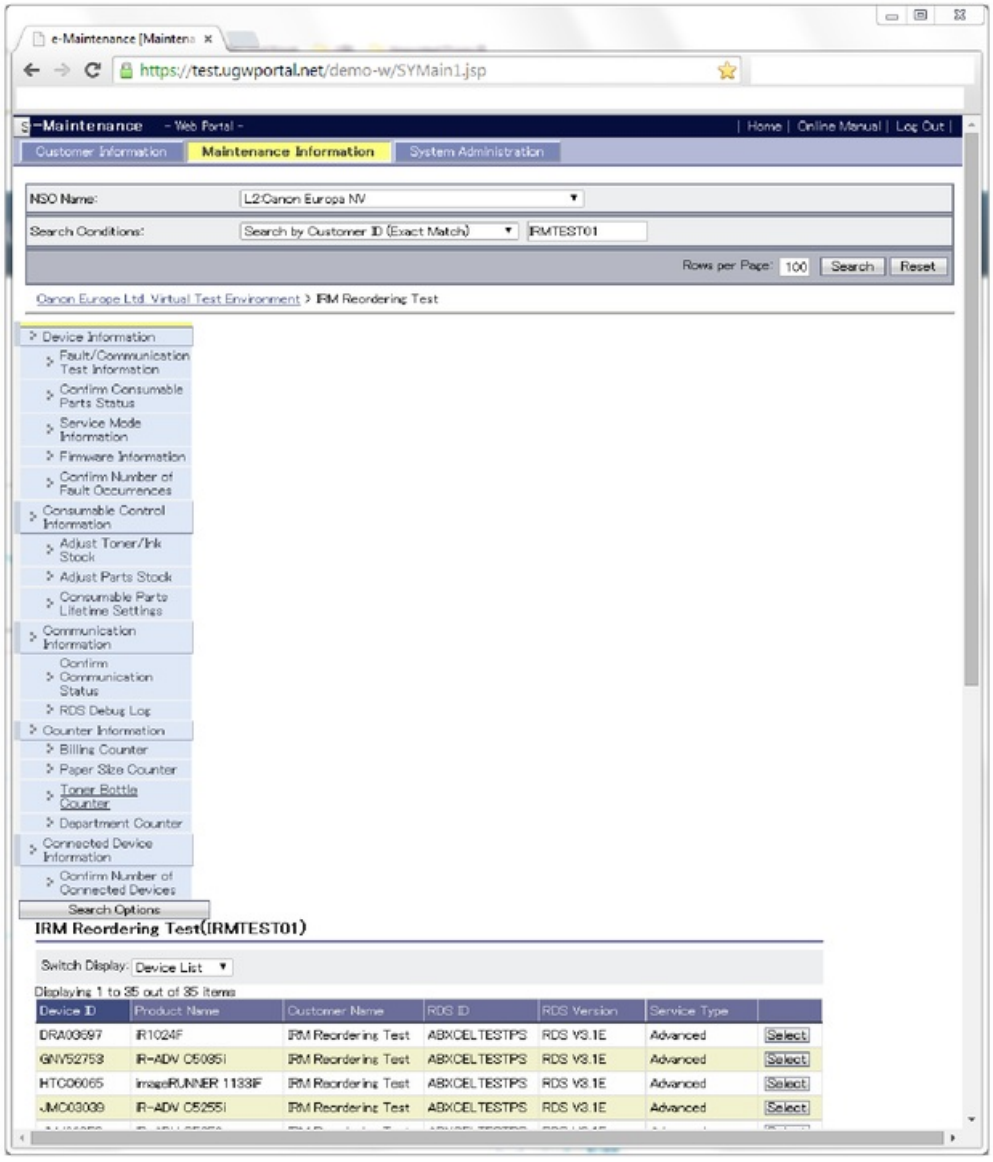
Whittaker, Z. (2011). Unlike Microsoft, Google can claim 99.9 percent cloud uptime. http://www.zdnet.com/blog/btl/unlike-microsoft-google-can-claim-99-9-percent-cloud-uptime/59104.

Xu, Q., Erman, J., Gerber, A., Mao, Z., Pang, J., and Venkataraman, S. (2011). Identifying diverse usage behaviors of smartphone apps. In *Proceedings of the 2011 ACM SIGCOMM conference on Internet measurement conference*, pages 329–344. ACM.

Yang, S., Yan, D., and Rountev, A. (2013). Testing for poor responsiveness in android applications. In *Engineering of Mobile-Enabled Systems (MOBS), 2013 1st International Workshop on the*, pages 1–6. IEEE.

Appendices

APPENDIX A



The eMaintenance website

APPENDIX B

Ability to...

Login to the test and production servers.
Handle forgotten and expired passwords.
Check and handle an account with a password that will expire within 10 days.
Check for internet connectivity and handle this appropriately.
Search for a specific device by entering a device ID.
Traverse a hierarchy of companies, customers and devices, and to filter these lists.
View a list of companies that are sorted according to the master company of each item, as well as the category of the root company.
View detailed device information.
View the firmware version of modules installed on a printer.
View the consumption rate of parts installed on the printer.
Check and handle the date of last communication between a printer and the servers.
Trigger the iPhone's email and phone applications upon selecting respective administrator contact details.
Display the number of events (errors, jams and alarms) that have occurred on a printer in the past 24 hours.
Provide a choice between the time frame in which events are retrieved, of 24 hours, 7 days and 3 months to be retrieved upon selection.
View a list of events and their handling status.
Refresh the list events to retrieve events that occurred between the previous time of retrieval and the current time.
View detailed information about events.
View information regarding the handling status of events.
Update the handling status of events by selecting from a choice of statuses and filling an optional comments field.
View a description of error and alarm events.
View the online FAQ of error events.

Functional requirements (in no particular order)

Stability
Usability
Maintainability
Responsive
Efficiency
Reliability
Fault tolerant (e.g. poor connectivity)
Scalability
Extensibility (introduced by developer)

Non-functional requirements (in order of priority)

APPENDIX C

```
//
//  ErrorHandler.h
//  eMob v9
//
//  Created by Emir Canpolat on 22/08/2014.
//  Copyright (c) 2014 CanonEurope. All rights reserved.
//

#import <Foundation/Foundation.h>
@class ArrayOf_ResultDetailType;

typedef enum ErrorType : NSInteger {
    IncUsrOrPsw,
    AccountLocked,
    PasswordExpired,
    SessionDisconnected
} ErrorType;

@interface ErrorHandler : NSObject

+
    (ErrorType)getErrorType:(ArrayOf_ResultDetailType*)arrayOfResultDetail;
+ (NSString *)getErrorDescription:(ErrorType)errorCode;

@end
```

```
//
//  ErrorHandler.m
//  eMob v9
//
//  Created by Emir Canpolat on 22/08/2014.
//  Copyright (c) 2014 CanonEurope. All rights reserved.
//
```

```
#import "ErrorHandler.h"
#import "ResultDetailType.h"
#import "ArrayOf_ResultDetailType.h"

@implementation ErrorHandler

// gets the error type of the error.
+
    (ErrorType)getErrorType:(ArrayOf_ResultDetailType*)arrayOfResultDetail
    {

    ResultDetailType* resultDetail = [arrayOfResultDetail
        objectAtIndex:0];

    int errorCode = [resultDetail.errInf integerValue];

    ErrorType errorType;

    switch (errorCode) {
        case 20001:
            errorType = IncUsrOrPsw;
            break;
        case 20002:
            errorType = IncUsrOrPsw;
            break;
        case 20003:
            errorType = PasswordExpired;
            break;
        case 20004:
            errorType = AccountLocked;
            break;
        case 20005:
            errorType = SessionDisconnected;
            break;
        default:
            errorType = errorCode;
            break;
    }

    return errorType;
}
```

```
// Get the error description according to the given error code.
// Return NSString*
+ (NSString *)getErrorDescription:(ErrorType)errorType {

    switch (errorType) {
        case IncUsrOrPsw:
            return NSLocalizedString(@"Incorrect_username/password",
                @"incorrect username/password error message");
            break;

        case AccountLocked:
            return NSLocalizedString(@"Account_locked", @"account locked
                error message");
            break;

        case SessionDisconnected:
            return
                NSLocalizedString(@"Not_logged_in_or_session_disconnected",
                @"Session disconnected message");
            break;

        default:
            return [NSString stringWithFormat:@"%d", errorType] ;
            break;
    }
}

@end
```

The header and implementation file of the ErrorHandler class.

APPENDIX D

```
// HomeTabTableViewController.m
#import "LargeActivityUIView.h"
...
_loadingView = [[LargeActivityUIView alloc] init];
...
[_loadingView setLabelText:NSLocalizedString(@"Logging_out", @"")];
[self disableUserInterface];
...
- (void)disableUserInterface {
   ...
    [self.view addSubview:_loadingView];
   ...
}
...
```

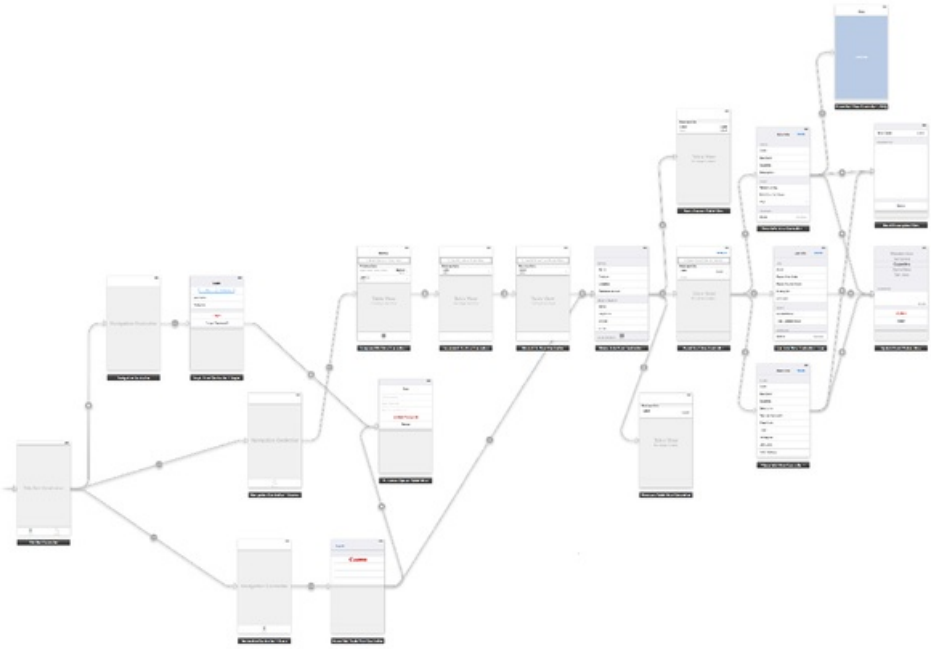
The loader is initiated and then its text is set before it is used. The NSLocalized-String is the internationalisation support of the application.

APPENDIX E

```
// CustomerIDsViewController.m
#import "AlertView.h"
#import "ErrorHandler.h"
...
ErrorType errorType = [ErrorHandler getErrorType:response.getRsp.
    getResDet];
[AlertView showAlert:NSLocalizedString(@"Error_text", @"error message")
    :[ErrorHandler getErrorDescription:errorType]];
...
```

The error type is retrieved using the error handler class (Appendix C). The alert is displayed using the error code.

APPENDIX F



The Storyboard at the end of the application.

APPENDIX G

```
// CompanyIDsViewController.m
...
// Loops through the company heirarchy dictionary (of keys) and adds
    the corresponding company object to a sorted and flattented company
    hierarchy of Company objects.
- (void)loopAndAddCompanies:(NSDictionary*)dictionary {

    NSArray* sortedDictionary = [self getSortedArray:dictionary];

    for (NSDictionary *company in sortedDictionary) {

        if (_once) {
            _rootCategory = [[_companyObjects objectForKey:[company
                objectForKey:_key]] getCompanyCategory];
            _once = FALSE;
        }

        if ([_rootCategory isEqual:[[_companyObjects
            objectForKey:[company objectForKey:_key]]
            getCompanyCategory]]) {
            [_flattenedCompanyHierarchy addObject:[_companyObjects
                objectForKey:[company objectForKey:_key]]];

            NSDictionary* nextLevel = [company objectForKey:_dictionary];

            [self loopAndAddCompanies:nextLevel]; // recursion
        } else {
            // do nothing
        }
    }
}
...
```

The error type is retrieved using the error handle. The alert is displayed using the

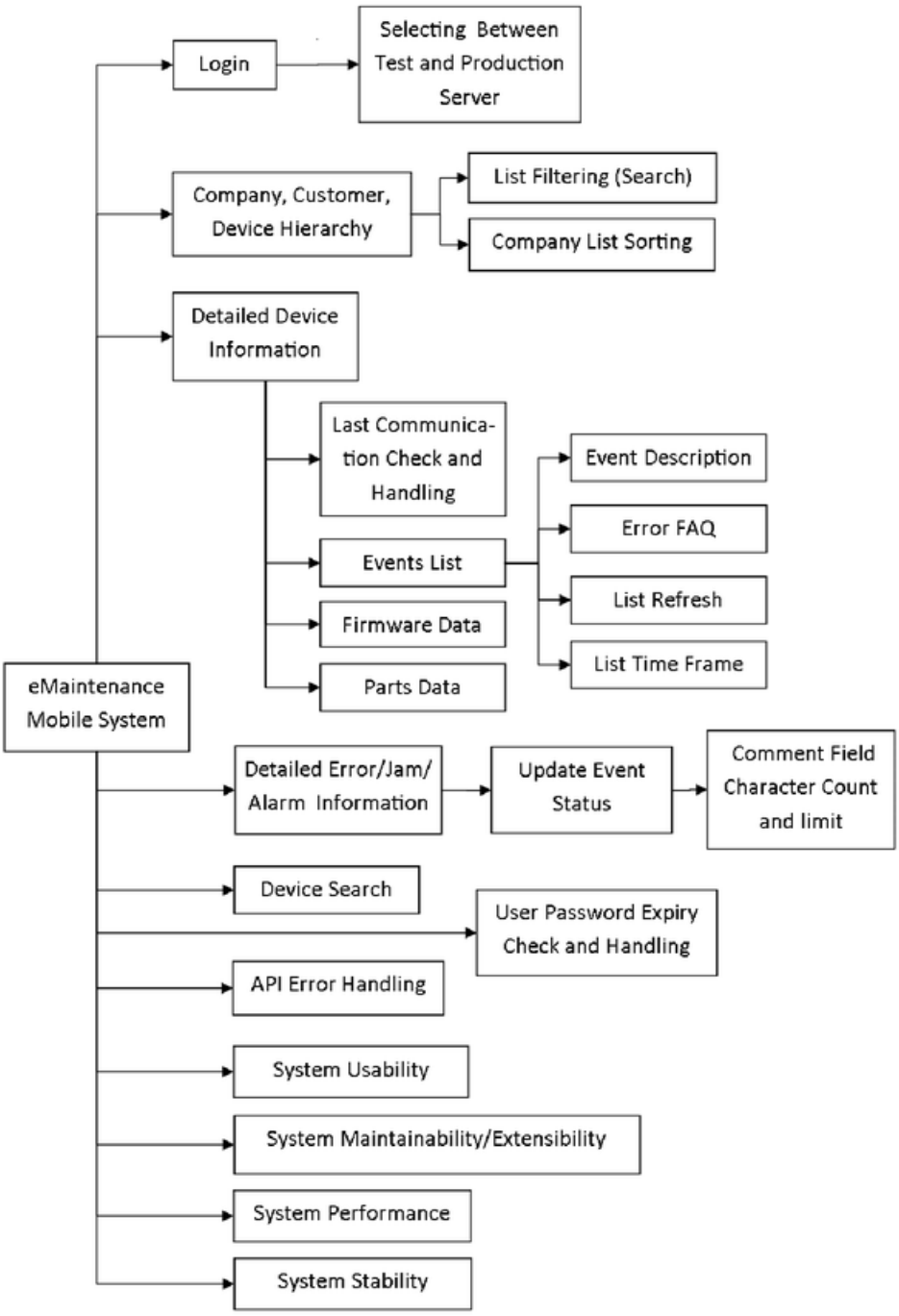
error code.

APPENDIX H

To login to the application enter:

Username: mobileappios
Password: password1234

APPENDIX I



The critical path of the development tasks

Emir Canpolat - ec363 - Dissertation

GRADEMARK REPORT

FINAL GRADE

/100

GENERAL COMMENTS

Instructor

PAGE 1

PAGE 2

PAGE 3

PAGE 4

PAGE 5

PAGE 6

PAGE 7

PAGE 8

PAGE 9

PAGE 10

PAGE 11

PAGE 12

PAGE 13

PAGE 14

PAGE 15

PAGE 16

PAGE 17

PAGE 18

PAGE 19

PAGE 20

PAGE 21

PAGE 22

PAGE 23

PAGE 24

PAGE 25

PAGE 26

PAGE 27

PAGE 28

PAGE 29

PAGE 30

PAGE 31

PAGE 32

PAGE 33

PAGE 34

PAGE 35

PAGE 36

PAGE 37

PAGE 38

PAGE 39

PAGE 40

PAGE 41

PAGE 42

PAGE 43

PAGE 44

PAGE 45

PAGE 46

PAGE 47

PAGE 48

PAGE 49

PAGE 50

PAGE 51

PAGE 52

PAGE 53

PAGE 54

PAGE 55

PAGE 56

PAGE 57

PAGE 58

PAGE 59

PAGE 60

PAGE 61

PAGE 62

PAGE 63

PAGE 64

PAGE 65