



Design Document

2012-13

Project Name: Geocaching

Team Number: 3

James Camden

James King

Emma Nugee

Alice Smiddy

Charles Wilson

0 Document Information

0.1 Version History

Ver. No	Ver. Date	Revised By	Description
1.0	21/01/2013	Alice Smiddy	Design Document Created
1.1	30/01/2013	Alice Smiddy	Created Introduction
1.2	05/02/2013	Alice Smiddy	Created Interprocess Dependencies
1.3	19/02/2013	James King	Created Intermodule Dependencies
1.4	21/02/2013	Emma Nugee	Created Interface Descriptions
1.5	27/02/2013	Alice Smiddy	Interprocess Dependencies draft complete
1.6	25/02/2013	Alice Smiddy	Introduction initial draft complete
1.7	28/02/2013	James King	Intermodule Dependencies draft complete
2.0	28/02/2013	All	Initial draft complete
2.1	04/02/2013	Emma Nugee	Introduction complete
2.2	04/02/2013	Alice Smiddy	Interprocess Dependencies complete
2.3	04/02/2013	James King	Intermodular Dependencies complete
2.4	04/02/2013	Emma Nugee	Interface Descriptions complete
2.5	5/02/2013	James King, Emma Nugee	Initial LaTeX typesetting complete
3.0	6/02/2013	All	Final document complete
3.1	7/02/2013	All	Final document run-through complete

0.2 Changes to Requirements

2.9 - Cache Scouting

Type	Functional
Description	If a user physically visits the location of a cache owned by a different user, they can choose to use points to scout a cache; and if the scout is successful, find out the point balance of that cache.
Priority	Medium
Pre-conditions	2.4 Cache Balance, 2.1 Account Balance, 2.2 Account Transactions
Input	The location of the user and the number of points they wish to scout with are supplied.
Operations	The distance of the user is checked to ensure they are sufficiently near the cache and the number of points used to scout is checked to ensure the user has at least that number of points in their account. The server will decide if the scout is successful.
Expected Results	If the request was valid and the scout action was successful, the user is informed of the number of points in the cache being scouted.

0.3 Table of Contents

Contents

0	Document Information	1
0.1	Version History	1
0.2	Changes to Requirements	1
0.3	Table of Contents	2
1	Introduction	3
1.1	Purpose	3
1.2	Scope	3
1.3	Definitions, Acronyms and Abbreviations	4
1.4	References	5
1.5	Overview	5
2	Interface Descriptions	6
2.1	Design Methodology	6
2.2	Process Descriptions	7
2.3	User Interface Design - Fortitude Application	10
2.4	User Interface Design - Fortitude-game.co.uk	15
3	Element Descriptions	19
3.0	Architecture Overview	19
3.1	Intermodule Dependencies	19
3.2	Interprocess Dependencies	27

1 Introduction

1.1 Purpose

The purpose of this document is to provide a thorough guide for developers to implement the virtual geo-caching application Fortitude. It expands on the higher level descriptions provided in the requirements document, through the use of clear diagrams and high and low level designs, so it can be used to create software that fulfills the specification laid out therein.

From this document a mobile application and supporting website will be produced that encapsulate an exciting game based on creating, defending and conquering caches. The system realised will allow users to create accounts that can be used for playing the game Fortitude, and will be able to interact with this game through an easy to use and appealing interface.

1.2 Scope

The design of the system covers user creating accounts to be used with the game, in which an army is utilized to conquer and defend caches (playing the role of forts). The aim of the game of the game is to own the most caches. The user's army size will grow relative to the number of caches owned, with the size of the garrison defending the cache influencing how often new soldiers are produced. Army size can also be increased by attacking outlaw camps which will not change ownership, only provide soldiers when defeated. Users gain caches by placing new ones and using their army to conquer other user's caches. They must leave soldiers at caches to defend against attacks. Game play happens through the application; the website allows users to manage accounts and plan strategies. In future releases, the website will include a forum or message board to further user interaction which allows for more complex strategies and alliances. In the current version, users can communicate via a private messaging system on the app and website. Administrators use the website to place or delete caches remotely, delete user accounts and act on user requests including questions or reporting a cache, a particular user communication or another user.

Accounts are created with a username, password and email address through the phone application or website, and are validated through an activation email sent to the user. Users can request a password reset email and, if their account is not activated, a new activation email. Once authenticated, the user builds up their army of soldiers and interact with caches at their location. Caches, the user's location are displayed on a map as well as a route to a cache if requested. Interactions with caches include placing one if 300m away from any other cache; adding or withdrawing soldiers from a cache the user owns to change the size of the defensive army guarding it; adding soldiers to an empty, unowned cache in order to become the owner; or scouting and/ or attacking an enemy cache. At any time, users can see basic information about caches, such as who owns it, and can the profile of that user, which provides information such as the number of caches they own.

User's scout a cache to find out the size of its defensive army without attacking it. Scouting is done by sending a number of soldiers to the cache(each of which can fail and die, which reduces the user's army). A user can attack a cache after or without scouting, and specifies the number of soldiers to attack with. If they are victorious they become the new owner of that fort, the surviving soldiers of their attacking army become the new defending army of the cache, and a proportion of soldiers originally defending the cache surrender and join the attacking army. If the attackers lose, every attacking soldier is lost, the surviving defenders will continue defending the cache, a proportion of the lost soldiers will surrender and join the army of the owner of the cache. After the battle, the user will be displayed a breakdown of results including the initial number of soldiers in each army, the winner of the battle, the survivors of the attacking side and the amount of soldiers that surrendered to the winning side. The user whose cache had been attacked will

receive a similar report through a notification system in the app, which will also alert users to messages received from other users.

Outlaw camps are non-player caches which can never be conquered, but are attacked like user owned caches. The camps are defended by a number of soldiers proportional to the total number of soldiers the attacking user owns, including defenders of the user's caches. Successfully defeating an outlaw camp, will give the user more soldiers than they lost in the attack, but if they lose then all the soldiers will be lost. After defeating an outlaw camp that user cannot attack the camp again for an amount of time that is proportional to the number of soldiers defending it.

Special event caches (or treasure areas) are very different from other caches as they are not visible on the map. When a user enters the range of a wireless access point with a MAC address specified by admins, the cache is found. Minimum distance rules do not apply to these caches because they are specified by MAC address and not GPS location. The caches reward users finding them with soldiers, however in future releases the treasure might include weapons or advanced soldiers that would strengthen the user's army. Each special event cache will only reward a user once and has a limited supply of treasure, so only a certain number of users can find it unless refilled by an administrator.

On the website, users can view caches on a map using a filter for example to search only for enemy caches. They can view information about their own caches and account including a record of their game history. A user can manage their account through the website, including updating their details, changing their password or requesting their account be deleted.

The application and website use a database storing each cache and user which is accessed and updated by a server. This server provides the functionality for the website and app, and checks the validity of data and commands to reduce errors. It also reduces cheating by ensuring location specific actions are performed at the correct location. Certain restricted features, such as promoting a user to an administrator, can only be accessed directly through the server.

1.3 Definitions, Acronyms and Abbreviations

GUI: Graphical User Interface. A GUI is a visual way of interacting with a computer via windows, icons, and menus.

View: A view is a superclass of all GUI element in the Android screen display environment

MAC Address: Media access control address. A MAC address acts as a unique identifier assigned to a network interface for communications on the physical network.

GPS: Global Positioning System. GPS is a space-based satellite navigation system that provides location , anywhere on Earth where line of sight is available to a minimum of four GPS satellites.

Database: A database is a collection of data organised to model relevant aspects of reality while allowing process requiring this information to access it.

Server: A server is a physical computer that is dedicated to run one or more services as a host. The client-server architecture implies the server is running to server the requests of other programs.

Splash Screen: A splash screen covers the entire screen and displays an image so the user has something to view while a program loads.

HTTP: Hypertext Transfer Protocol. HTTP is the foundation of data transfer for the World Wide Web, it is an application protocol for distributed, collaborative, hypermedia information systems.

API: Application Programming Interface. An API protocol is intended to be used as an interface by software components to allow inter-component communication. An API library may contain specifications for routines, data structures, object classes, and variables.

DBMS: Database management system. A DBMS program is a program that enables storing, modifying, and extracting information from a database.

.NET: The .NET Framework is a software framework developed by Microsoft which includes a large library and provides language interoperability across several programming languages.

CLR: Common Language Runtime. The CLR is the virtual machine component of Microsoft's .NET framework and is responsible for managing the execution of .NET programs.

SQL: Structured Query Language. SQL is a programming language designed for managing data held in a relational database management system.

SQLite: A relational database management system.

IDE: Integrated development environment. An IDE provides facilities for software development via a software application.

SqlCe: SQL Server Compact. SQL CE is a compact relational database for applications that run on mobile devices and desktops.

DDL: Data definition language. A DDL is similar to a computer programming language for defining data structures.

LINQ: Language Integrated Query. LINQ is a .NET Framework component that adds .NET data querying capabilities.

JSON: JavaScript Object Notation is derived from the JavaScript scripting language designed for human-readable data interchange.

HTML: HyperText Markup Language. HTML is the main markup language for creating web pages and other web browser based content.

PHP: PHP is a server-side scripting language designed for Web development, to allow for production of dynamic Web pages.

CGI: Common Gateway Interface. The CGI is a standard for executable files known as CGI scripts to generate web content when delegated by web servers.

AJAX: Asynchronous JavaScript and XML. AJAX is a group of interrelated web development techniques used on the client-side to create asynchronous web applications.

1.4 References

23 Cats, 2012. *Requirements Document 2012-13, Project Name: Geocaching*, Unpublished

Bell, D., 2005. *Software Engineering For Students: A Programming Approach*, 4th edition, London: Pearson Education Limited

Dix, A., Finaly, J., Abowd, G., Beale, R., 1993. *Human-Computer Interaction*, Cambridge: Prentice Hall International (UK) Limited

Sommerville, I., 2004. *Software Engineering*, 7th edition, New York: Addison-Wesley Publishers Limited

1.5 Overview

The remainder of this document will be a thorough guide for developers of this system. It will be split into Interface Descriptions and Element Descriptions. In the Interface Description section there will be mock ups of the user interface for both the app and website and descriptions and diagrams of how a user might use these. The Element Description section will design the system architecture and core parts of the system, and explain decisions along the way. It will do this through design elements each with a specific function, shown through block diagrams. Class diagrams will be used to show the static structure of the system and dependencies between components. Activity diagrams and sequence diagrams will be used to show the dynamic behaviour of the system.

2 Interface Descriptions

The design of a good user interface is key to a successful system, playing a vital role in attracting users. Indeed Douglas Bell (2005:53) stated that the user interface is the "yardstick by which a system is judged". This section will describe how the final product will be suitable, appealing and fulfil the criteria in the requirements document (23 Cats, 2012).

2.1 Design Methodology

2.1.1 Fortitude Application

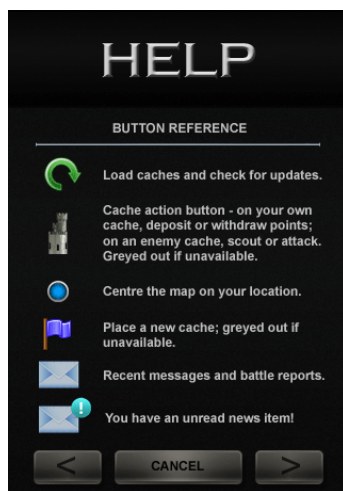


Figure 2.2: Button reference in the help section.

Our design is easy to use for users of various levels of familiarity with the system. The domain analysis highlighted the significance of intuitiveness and easy accessibility (23 Cats, 2012:5-7); therefore the interface has been designed to be clear and informative including help screens where appropriate (see fig 2.2). A well designed user interface reduces errors (Sommerville, 2004:363-66) so many features of the app have been tested on potential users during development. In particular, it had been decided to disable the android back button to give the implementers greater control over which screens the user could access at any point in time. This proved difficult for users to adapt to, so instead the back button was incorporated into the application.

To further ease of use, buttons will be greyed out when their action is not available and will perform no function. For example when a user is too close to an existing cache the 'place cache' button will be greyed out (see fig 2.3).

The application's screens satisfy requirement 5.3.7: Interface Style Uniformity by their colour scheme and placement of buttons and headers. The colour scheme is silver or white on black, as this is easily read, even by people who are colour blind. The app provides a fast response to user input by using separate images for on-press actions ensuring the system has received the user's request (see fig 2.4), which satisfies requirement 5.3.9: Interface Feedback Delay.



Figure 2.3: An inactive button (left) and an active button (right).

2.1.2 Fortitude-game.co.uk Website

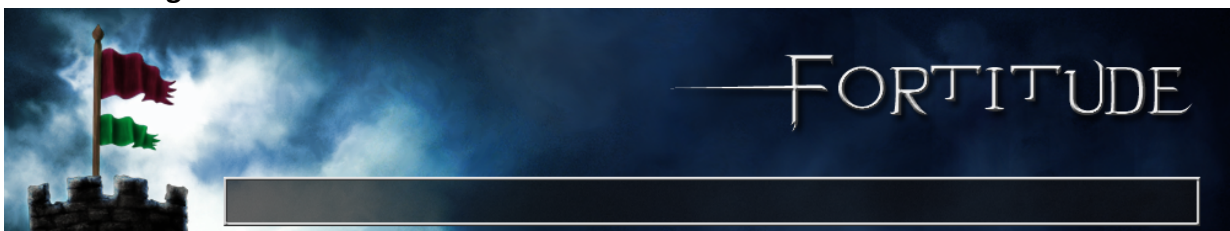


Figure 2.4: The header image, an example of the more complex graphics used in the website. The white bordered box will hold the navigation.

The website is a supporting feature to the Fortitude application. With its larger dimensions and lack of battery and data restraints, the website can provide functionality not feasible with the application, such as requirement 5.3.6: Activity Recording. This means the app can be simpler, and therefore more user friendly without compromising the scope of the system. Through the website, administrators can interact with app as specified by many requirements including 5.1.3: Administrator Accounts. This is an easy to access alternative to bespoke software, and is not device dependant so administrators can perform all actions without an Android device. The website design matches the application design by using the same colour scheme, recognisable logo of two flags flying over a castle tower and a navigation menu that echoes the message box of the app, which satisfies requirement 5.4.8: Website Application Style Uniformity. However the design has been expanded upon in the website to be more visually appealing, while these graphics could not be included in the app without sacrificing ease of use.

Commonly accessed areas of the website, such as the sidebar and the homepage, are organised into modules (see fig 2.5), making information quickly and easily identifiable. It also provides an excellent template for adaptation; though out of scope for this project, users could in future personalise the website by selecting which modules to display.

The website provides feedback, by validating data (see 2.6) or by providing loading messages, so the user can interact efficiently and with fewer errors. For more critical data entry, such as an administrator creating a new cache, the user is presented with a summary page of the information they have entered and must confirm it is correct before the new data is stored.

The website also provides feedback if an error occurs - the most common anticipated is the user having javascript disabled. This will cause every javascript dependent element of the site to be replaced with a placeholder of the correct shape to avoid disturbing the website layout which informs the user that javascript is required.

2.2 Process Descriptions

2.2.1 Opening the application

The process of opening the application will be quick and efficient. Therefore the user will be taken straight to the main screen from the splash screen if they are already signed into the application.

Users who have not activated their accounts are given limited functionality while using the application, which fulfils requirement 5.1.2: Account activation. The screen to create a new user leads directly to the unactivated main screen; and the user can get to the activated main screen when they open the app after activating their account through the verification email sent to them. As per requirement 5.1.4: Resend activation email, the user can request a new activation email from the unactivated main screen.

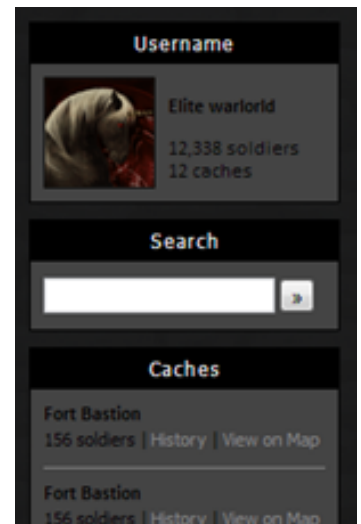


Figure 2.5: Examples of the individual 'modules' that comprise the website (in this instance, the sidebar).

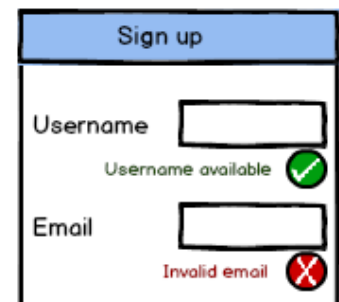


Figure 2.6: Wireframe showing user feedback when creating an account.

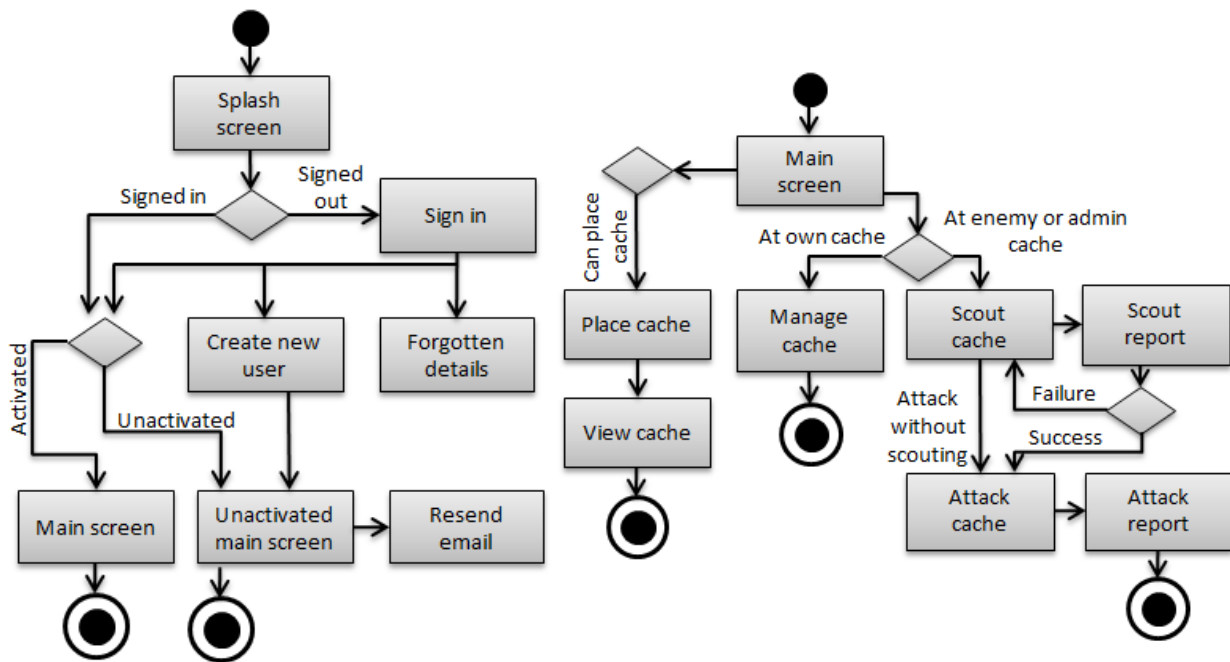


Figure 2.7: Screen map for the process of opening the app.

Figure 2.8: Screen map for key game actions.

2.2.2 Performing key game actions

The key actions to Fortitude game play are creating caches (place cache), defending caches (manage cache) and conquering caches (scout then attack cache), which are all accessible through the main screen. These actions require the user to be at the cache or cache they want to place, otherwise the buttons will be greyed out (see fig 2.3). The same button is used to manage a cache (deposit or withdraw defending troops, or abandon the cache by withdrawing all troops) and attack an enemy cache. This simplifies the process of playing the game for the user and reduces redundancy and clutter on the main screen.

2.2.3 Viewing caches

Viewing caches is integral to the above actions and can be done from the main screen (see fig 2.9) whether or not the user is at the cache. This allows the user to plan strategies, keep track of their caches and plan routes to caches displayed on the map (see fig 2.14), which satisfies requirement 5.3.4: Path finding. The user can also view a profile of an enemy through their cache and report the cache to admins (see 2.2.5).

After a user's cache is attacked, they can see the battle report from their notifications and then view the cache. If their defenders were victorious, they will see an allied cache, but if the attackers won they will see an enemy cache.

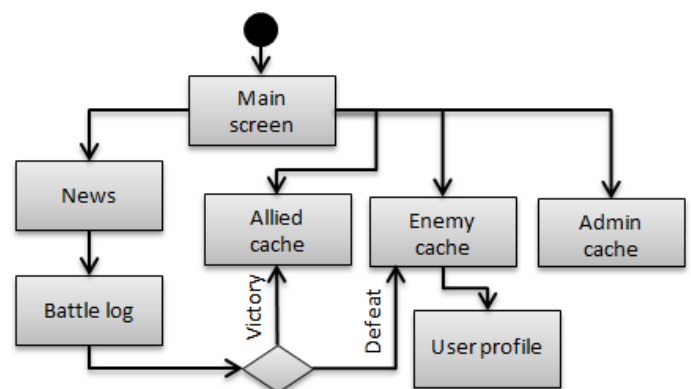


Figure 2.9: Route mapping.

2.2.4 Reading and sending messages

Users can interact with each other by sending messages, allowing alliances and battle strategies to be formed and satisfying requirement 5.4.1: User communication. From the inbox, users are given the option to turn the messaging system off. If, due to this or any other reason, a user can not send a message then they will be informed through a 'Failure' screen. This is so a user is informed and won't send multiple messages believing the previous ones have failed. Users can also report communications and block users (see 2.2.5).

Requirements 5.2.2: Cache reporting and 5.4.2: Communication reporting both require the user to be able to alert the administrators of certain caches or communications. This is anticipated to include a cache placed on a private or otherwise inaccessible location, or users behaving inappropriately towards others, and will be handled through a single report form (fig 2.8). The subject line is automatically filled in according to the screen from which the user reported from (see fig 2.11), and a copy of the offending message, user profile or cache is sent with the body of the report that gives administrators as much information as possible.

Outlaw camp	Outlaw camp
Enemy cache	User owned cache
Allied cache	User owned cache
Message	User communication
User profile	User profile

Figure 2.11: Screens from which a user can report, and the corresponding subject line.

Blocking users stops the user who blocks from receiving messages from the users they have blocked. From their inbox, a user can see a list of blocked users, block and unblock users. Blocking can be done from the user profile too.

2.2.5 Administrators acting on user request

2.2.6 Reporting and Blocking

User requests from the app are always in the form of a report as detailed in 2.2.5. A user may submit other requests from the website, including a request to delete their account or a support question sent through the help page. All of these requests must be acted upon by an administrator through the admin section of the website. When a user deletes their account they must confirm their action through an email sent to them before the request is sent to the administrators.

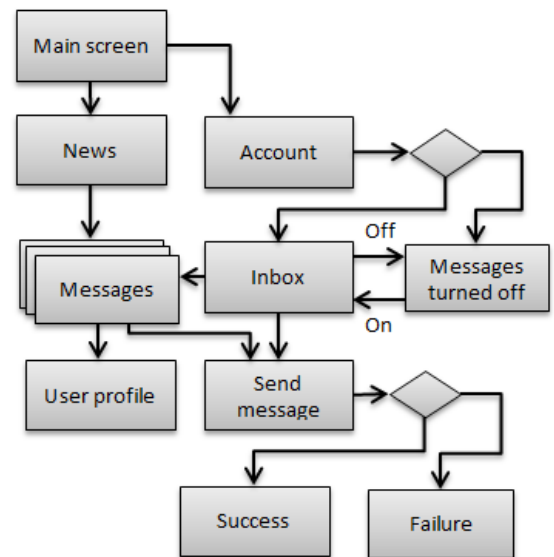


Figure 2.10: Screen map for the messaging system.

Figure 2.12: Send report.

On the website, administrators are able to see a page of all user requests not being worked on by another administrator. The administrator selects a request, which removes it from the requests page, and performs an action on it to complete the request or flag it if it can't be dealt with by that administrator at that time. The administrator will be informed of the result of their action and any details about a failure, such that the administrator can correct their mistake or flag the request. In the unlikely event that two administrators try to action the same request, the first action will be performed and the second administrator informed of this when they submit their own action. Once acted upon, requests will be accessible through a 'past requests' page.

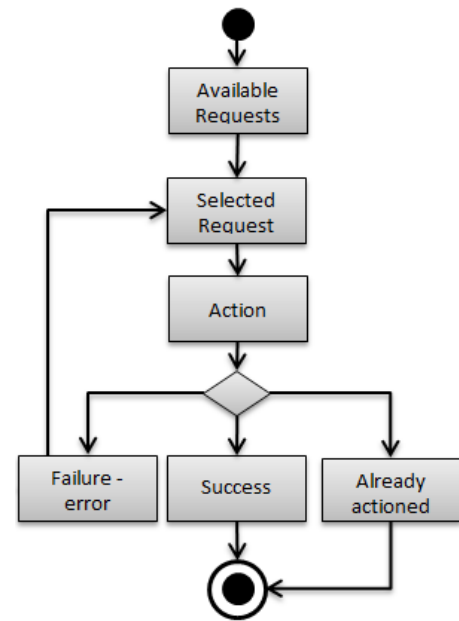


Figure 2.13: Administrators acting on a user request.

2.2.7 Miscellaneous other

A user can sign out from the user account settings, accessed from clicking on the status bar on the main screen.

Special placement caches appear on the screen if the user has the application open at the correct location. They can then choose to accept or reject the award for finding the cache.

Routes to caches are shown on the main map as a thick red line (see fig 2.14). They can be planned from 'view cache' screens, and cleared in the account menu. As routes are cleared before a new one is made only one route can be displayed on the map at once.



Figure 2.14: Displaying a route to a cache.

2.3 User Interface Design - Fortitude Application

2.3.1 Main screen

The main screen is the core of the Fortitude application through which all key features are easily accessible. These are split into features relating to playing the game, grouped in the action bar at the bottom of the screen, and features relating to providing the user with information or allowing them to access their account, grouped in the status bar at the top of the screen. When clicked, this status bar leads the user to their account menu through which they can sign out, access their message inbox, or view the help screen. The status bar also displays the point balance available to the user; as points are required for core actions, it is important that this information is very visible, as specified by functional requirement 5.2.1: Account balance.



Figure 2.15: The 'Recent news' icon and (right) with notification.

The action buttons relating to game play are grouped at the bottom of the screen, where it is most ergonomic for the user to access them. Each button is clearly distinct and, where appropriate, contains easily discernible extra information - such as the place cache and cache action buttons being greyed out when unavailable (see fig 2.3) and the recent news having a notification symbol to indicate unread news (fig 2.15).



Figure 2.16: The main screen of the Fortitude application.



Figure 2.17: The pins are easy to tell apart without colour.

The main area of the home screen is the map, chosen as a very user friendly interface for the game and fulfilling several requirements including 5.3.1: Display location and 5.2.3: Nearby caches. Caches are clearly distinguished with different coloured pins to differentiate between admin, allied and enemy caches, and allied caches further distinguished by the presence of a dot, as per requirement 5.2.3: Cache ownership and 5.2.26: Distinguishing cache owners; the colours are chosen to account for those with colour blindness (see fig 2.17). The map itself is controlled

with the touch screen, using swipe or drag to navigate and pinch to zoom (zoom required by 5.3.3: Map zooming). To reduce data usage as per non-functional requirement 5.2.30: Data usage, adjusting the map view by zoom or location will not load the caches for that area of the map. This will be done by the green refresh arrow on the right of the status bar which also updates point balance, a route to a cache if active, and recent news feed as required.

2.3.2 Unactivated main screen

The main screen for the unactivated user is identical in appearance to the main screen for an activated user (see fig 2.16), with the only difference being in the bottom bar. On the standard main screen this contains several buttons linking to features or actions which are not available to unactivated users - as specified by functional requirement 5.1.2: Account Activation. Here, this bar is replaced by a banner reminding users to activate their account; when clicked, this banner leads to the resend activation email page.

All other features have the same functionality as the main screen. The only difference is in navigating the map; on the unactivated account, each request to load a new view of the map also loads every cache within that view. This is because the unactivated user is not able to refresh the screen, as this button is located on the action bar which is not available here.

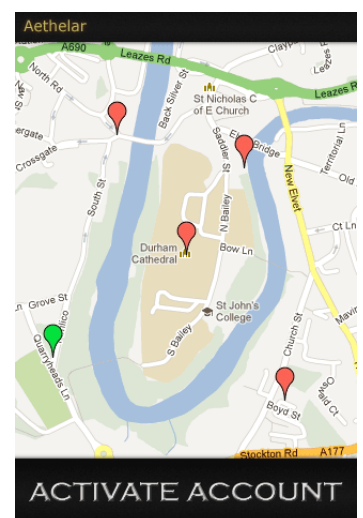


Figure 2.18: Unactivated main screen.

2.3.3 User text input

Many screens make use of text the user has input, most notably those involved with signing in or creating a new account (see fig 2.6) as well as sending messages to users (fig 2.10) or reports to administrators. Text input is handled through the on screen keyboard that is standard on an android phone (see fig 2.20).

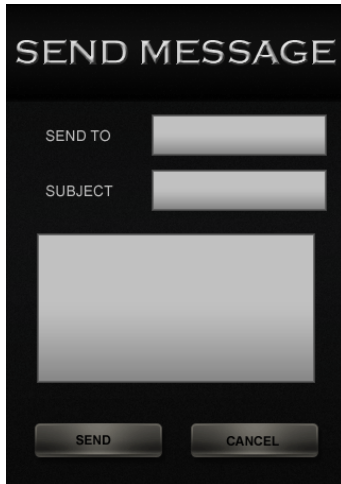


Figure 2.19: Send message screen, showing user text input fields.

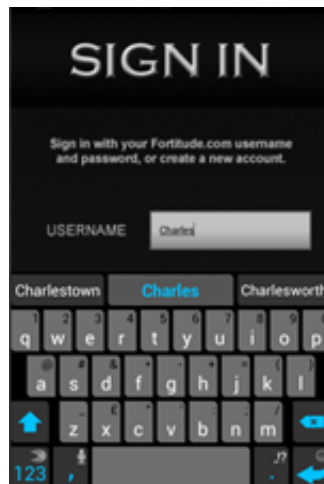


Figure 2.20: Using the on screen keyboard to sign in.

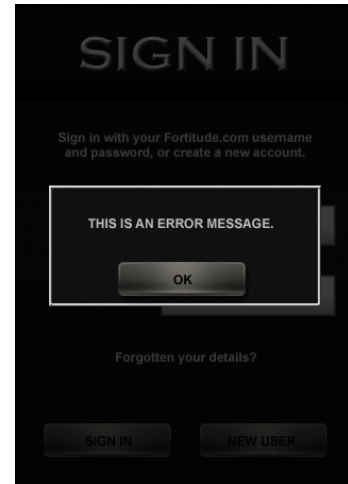


Figure 2.21: An example of a message box displaying an error message.

Should the user input invalid information, such as incorrect login details, then an error message will be displayed (see fig 2.21). The screen behind this message will be dimmed to ensure the message is clearly visible. A similar design will be used for all message alerts to the user, such as error messages, status messages (for example 'connecting to server') or confirmation messages, which may have two options for the user to choose between (see fig 2.22).

Message boxes and text input fields are always of a consistent design to make them easy for the user to identify, though message boxes may stretch to contain longer messages.

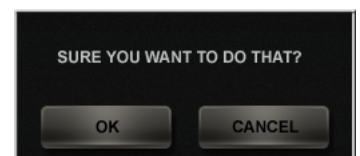


Figure 2.22: A message box with two options.

2.3.4 Menu and Information screens

Many screens in the application give the user a set of options or provide some form of information. Viewing caches is one such example, as is the account menu (fig 2.23). The primary aim of these screens is to make the information as clear as possible to the users while at the same time making efficient use of space, so that the user does not have to click through multiple nested menus to access any feature. There is also consistency across screens; the 'cancel' button, for example, is always either the central bottom button or the bottom right.

The use of images when viewing a cache provides not only aesthetic appeal, but also serves a functional purpose. With different images used for enemy caches, allied caches and administrator caches, it is easy to identify from the cache screen what type of cache is being viewed; again, this refers back to requirement 5.2.26: Distinguishing caches.

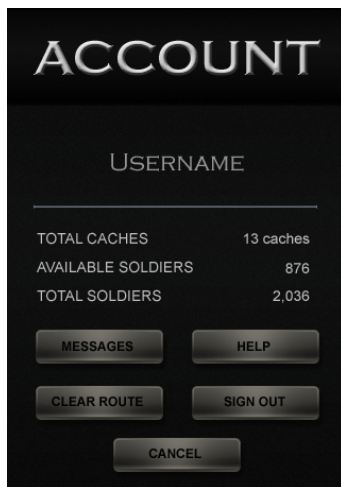


Figure 2.23: The user's account menu.

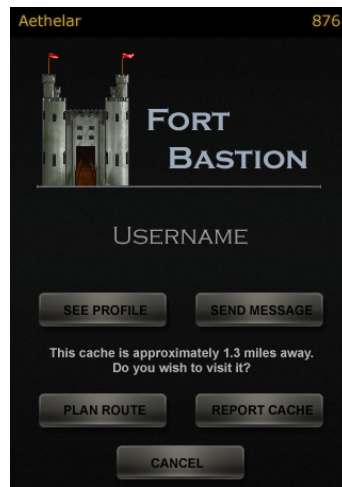


Figure 2.24: Viewing an enemy cache.

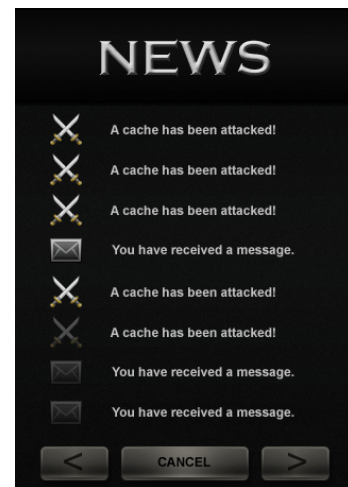


Figure 2.25: A page of items in the recent news.

2.3.5 List screens

Lists are used for displaying news items and messages in the inbox. Each screen displays several items on a single screen with forward and back buttons at the bottom of the screen to view the next page of items. This helps to limit data usage and loading times for the user whilst also providing an easy to navigate, well ordered system of displaying the items.

The use of icons to distinguish between types of items - battle reports and news icons, for example - makes it easier to take the list as a whole in without difficulty, and fading the icons of items which are not new helps the user find the important information with greater ease (as shown in the bottom 3 icons of fig 2.25).

2.3.6 Action screens

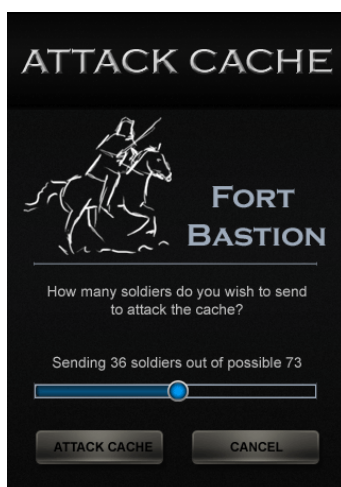


Figure 2.26: Attacking a cache.

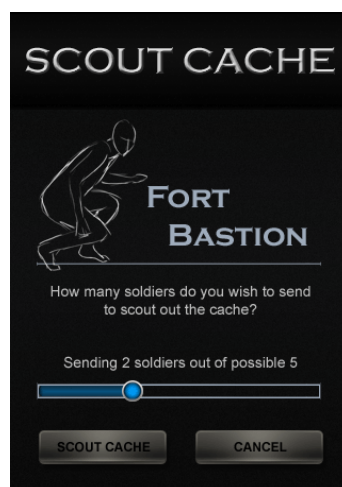


Figure 2.27: Scouting a cache.

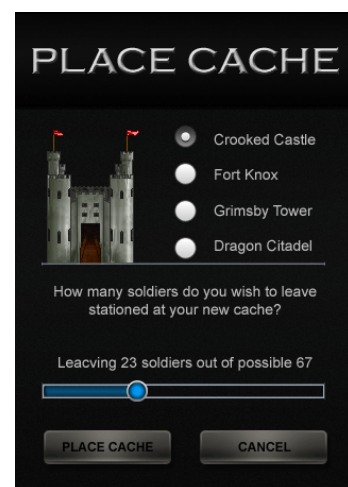


Figure 2.28: Placing a cache.

Screens involving direct interaction with caches, such as attacking, managing (withdrawing or depositing soldiers) or placing caches are of a similar layout to provide consistency throughout the game. The screens are clearly differentiated by the use of different images - such as a charging soldier on the attack screen versus a sneaking soldier on the scouting screen (see figs 2.29 and 2.30) as well as by the action title being clearly displayed. The name of the cache involved is also clearly displayed on the screen. Any time that user input is required in these screens it is done in such a way as to restrict the values of possible user input to those which are known to be valid, such as using a slider to change the number of soldiers committed to an action or choosing between a choice of names when creating a cache (see fig 2.28). This is to make the process of game play as quick and error free as possible.

2.3.7 Reporting results

Any action with an unknown outcome is reported to the user using a screen of the same design, again to make the app consistent and increase the user's familiarity with how it works and where information is displayed. Where relevant, the image of the report screen reflects the outcome - such as a scroll for a successful scouting mission versus a tomb stone for a failed one (fig 2.29). As ever, this is both for aesthetic reasons and to communicate the necessary information to the user clearly and quickly.

A similar report screen is also accessible from the news feed for the report of a cache the user owns which has been attacked by another player. This screen differs in that the name of the cache is more prominent and there is a link to view the cache on the map; this is required to identify the cache as the user may not be present at the cache's location.



Figure 2.29: Unsuccessful scout mission report.



Figure 2.30: Reporting a battle result.

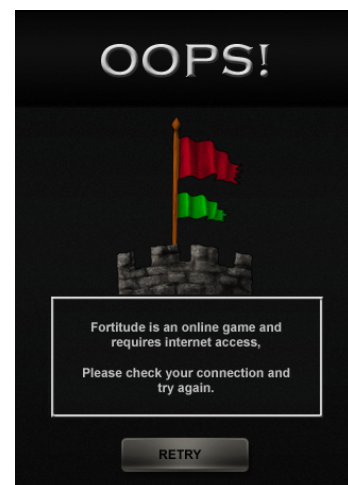


Figure 2.31: An error caused by lack of internet connection.

2.3.8 Static screen

The final type of screen is a static screen. This is one not designed for much user interaction; the full screen is comprised of a single image or static text, such as the Fortitude logo on the splash screen or the help screens (fig 2.1). The splash screen itself has no buttons for the user to progress, as the app will automatically move on once it has loaded. Other static screens, such as the help screens, have a button to move on from this screen.

Static screens are used for errors which cause the app itself to be unplayable - the most common of which would be the user not having internet connection available (see fig 2.31) - and the user will be unable to move from this screen until the error has resolved. A static screen is also used for when a user comes across a special placement cache, with which they cannot interact but must either accept or reject the prize.

With the exception of the help screens, static screens cannot be called by the user; they are initiated by the server and used to send a message to the user that cannot be ignored or minimised until normal game play can resume.

2.4 User Interface Design - Fortitude-game.co.uk

2.4.1 Site Layout

The visual design of the website was chosen to reflect the dramatic battle-driven nature of the game of Fortitude. It clearly echoes the app in its sombre colour scheme of black, white and greys, and the use of the two flags image used as Fortitude's logo (see fig 2.4). The choice of a more serious design both in the website and the app is to reflect the strategic and 'epic' style of game that Fortitude is, appealing to players of all ages.

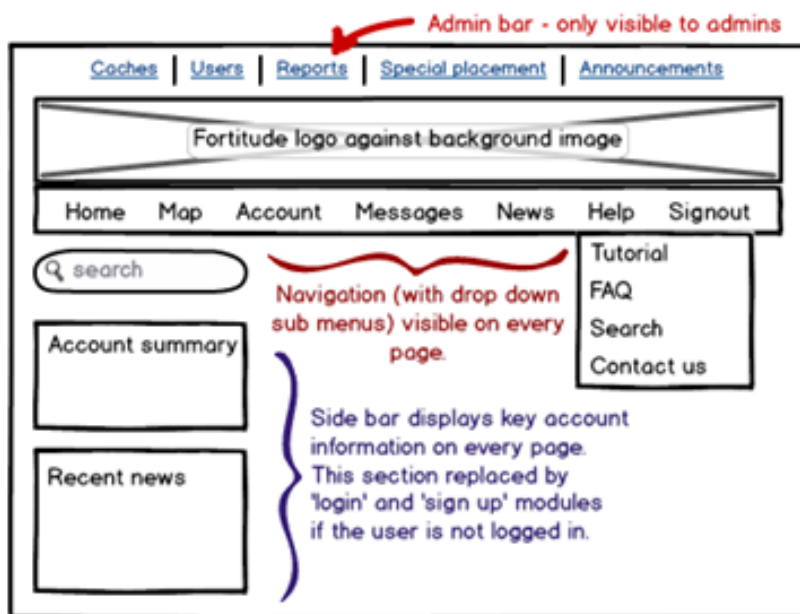


Figure 2.32: Wireframe of the site layout with space left for page content. The image used in the header is shown previously in figure 2.4 on page 6.

should they reach an administrator page, they will be shown a 404 page. Similarly, some pages are restricted to users; should a visitor who is not logged in attempt to view a page such as the account page, they will be redirected to the login/sign up screen.

The use of drop down sub menus is important to increase ease of navigation, enabling users to quickly reach areas of the website without over cluttering the top bar itself. However, as not all devices have a hover feature (such as tablets or phones), no area of the website will only be accessible through these sub menus.

The basic structure of the website is for the page to be divided into three, with the top area containing navigation (including administrator links), the sidebar to the left containing key information and the larger area to the right displaying the page content for each page (see fig 2.32). Both the sidebar to the left and the main page content of the home page use modules to display content, making the information easily and quickly accessible. The administration links have been placed above the main page to ensure that they cannot interfere with the usual page content but are not sidelined and hard to get to; it is anticipated that administrators would mainly use the website for admin purposes, and so need the links to be clearly visible. These links will not be displayed for any user who is not an administrator;

2.4.2 Site Content

The content of the website will in many cases mimic that of the app, though in greater depth or with greater functionality - for example, the user can view caches on a map as they can through the phone, but can also apply a filter to display only their own or only enemy caches, as per requirement 5.4.7: Overview Map (see fig 2.33). In addition, some functionality is only available through the website - such as the ability to search for a user or cache by name, or the ability to view an activity log of recent activity associated with that account, specified in requirement 5.3.6: Activity Recording. Certain content, such as the home screen and site news, is visible to all whether logged in or not so as to give potential users a feel for the game and encourage them to sign up and participate. Other areas will redirect the user to the Login page which consists of two modules, one to log in and one to sign up. These modules will also be visible on the side bar at all times unless the user is logged in, in which case the side bar will display key information.

2.4.3 The Map

The map will use the familiar google maps interface with zoom, street view, and different types of map (satellite, terrain etc). The user can search the map by location, username (ie all caches owned by a certain user) or cache name; should multiple caches with the same name be found, both would be displayed on the map (and the zoom adjusted until both were visible), and the user could zoom in on the correct one. The option to filter caches is done through a drop down menu with filters including 'All caches', 'My caches', 'Enemy caches', and 'Outlaw camps' (non-player caches). Updating the map by either filtering or searching is done asynchronously, providing a much smoother experience for the user without having to reload the page each time a request is sent.

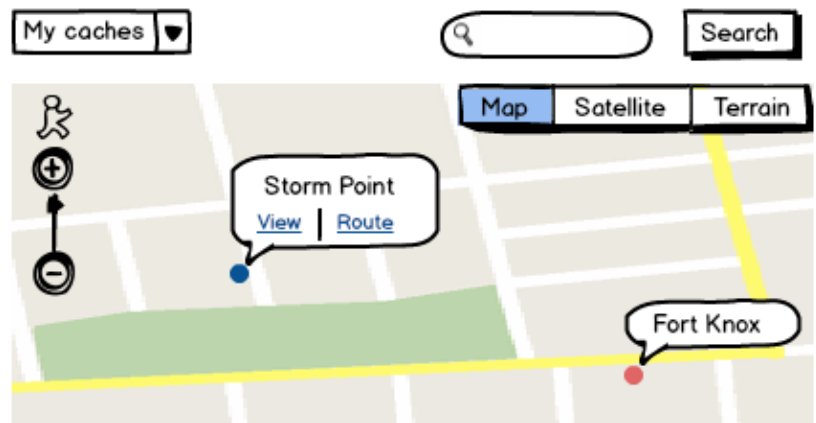


Figure 2.33: Wireframe of the map and controls

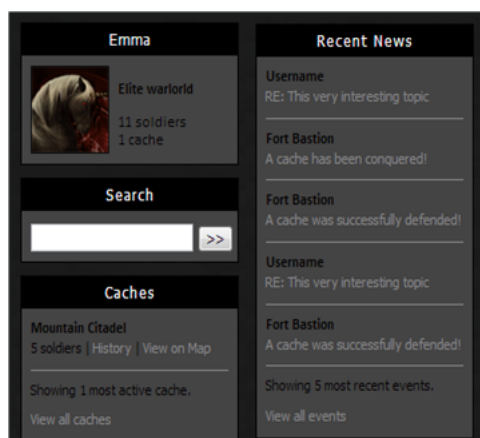


Figure 2.34: The key information displayed in the sidebar (recent news would be placed below rather than beside the other modules).

Different types of caches are distinguished as on the app by colour and the presence of a dot, such that colour blind people can also tell caches apart. A cache displays its name and, when clicked on, choices to view the cache details or plan a route to it; this option will bring up a dialogue box asking for the location to plan the route from.

2.4.4 Sidebar

As mentioned previously in the Design Methodology (2.1.2), users will in future be able to personalise sections of the website including the sidebar to display the modules most appropriate to them. Until such a point, the sidebar will display to a logged in user a search bar, an account summary, a list of the user's five most active caches - determined by the cache with the greatest frequency of events in its history - and a list of the user's five most recent

events. If a user who is not logged in views the site, the sidebar will contain the search module as well as modules to log in or sign up.

The title 'Elite Warlord' in fig 2.34 Corresponds to the user's level, determined by the number of caches they own. The avatar and user title serve little purpose as far as game play is concerned, but contribute towards personalising a user's account and, in the levels, giving a tangible goal to aim for in the game.

2.4.5 Administrator pages

The administrator features of the website are accessible through the navigation links at the top of the screen, which are only visible to users logged in as administrators. These are highlighted in green, the colour making them stand out and separate from the rest of the website. The tasks that can be completed from each page are shown below in fig 2.36.



Caches :: Users :: Reports :: Special Placement :: Announcements

Figure 2.35: The administrator links. The bar itself extends horizontally to fill the page.

Caches	View all caches, grouped under 'user cache' and 'outlaw cache'. Includes options to view the history of or delete caches of any kind or add outlaw caches, as well as managing or editing outlaw caches.
Users	View the entire user list. Includes options to send users an official warning or other communication and to remove a user account (this would only be done by user request).
Reports	View all user requests that need to be acted on, and select individual requests to action.
Special Placement	Create and manage special placement caches.
Announcements	Post an announcement to the site (appears on the home page and news), and edit or delete old announcements.

Figure 2.36: Tasks carried out by admins from each administrator page.

Administrator pages can be roughly grouped into two types; those that involve viewing a database, and those that involve creating data such as a cache, special placement or announcement. Database style pages are laid out as shown in fig 2.37 (the header and sidebar sections are not shown, but are the same as the rest of the site). Filtering the database and commands such as creating new caches or, on the user request page, viewing past requests are linked from the top of the page for easy access, and the database is grouped by a tab system into the types of database. This avoids having redundant fields in an entry - such as the 'owner' field for an outlaw camp entry. The administrator cannot ever see the whole database on one page; they are originally shown only the filter section and must access the database by running a query from here. Should the query be too large, the administrator will be requested to narrow down the filter.

Caches

Filter by Location ▼ [Apply filter\(s\)](#) | [Add new filter](#)

Show all within 3 kilometres of latitude longitude


☒ coordinates
☐ address

User owned
Admin owned
Outlaw camp

Create new cache

ID ▲	Name	Location	Founded	Owner	Notes	Remove
3445	Fort Bastion	(lat) (long)	12/03/2012	username		<input type="checkbox"/>
231	Dragon Palace	(lat) (long)	18/02/2012	username		<input checked="" type="checkbox"/>

Hovering over the location of a cache shows it on a map



Removing a cache deletes it from the database. Checking this box causes a dialogue box to appear confirming the action.

Figure 2.37: Wireframe example of a database page, here managing caches with a location filter selected.

Creating data is handled through a form interface which, despite its simple design, is arguably the best method of data input for users (Dix, Finlay et al, 1993:105). The form provides user feedback for incorrect or invalid data entries in the same way as creating a user account would (see section 2.1.2).

3 Element Descriptions

3.0 Architecture Overview

It was apparent from the briefing and requirements that the most suitable architecture for the system would be a client-server model. There will be a central server that stores persistent system related data, and performs the majority of the application logic. This server services requests from a client application that handle user interaction and presents information through a graphical user interface. Another requirement (5.4.3) was to provide a website as a second source of information and user interaction, which will be implemented using the central server.

The server and client may be conceptually divided into several modules. For the server, there is the *Database Module* that manages reading from and modifying the persistent application data, the *Logic Module* that handles game and system calculations, and the *Web Interface* group which contains the *API Module* for interacting with client applications, and the *Website Module* for servicing the website HTTP requests. The client will be constructed to include a *Request Module* for contacting the server and interpreting any responses, a *Logic Module* that is aware of relevant aspects of the game state and performs client-relevant calculations, a *Geolocation Module* for interaction with the Google APIs and the GPS sensor, and a *Window Module* for the graphical user interface.

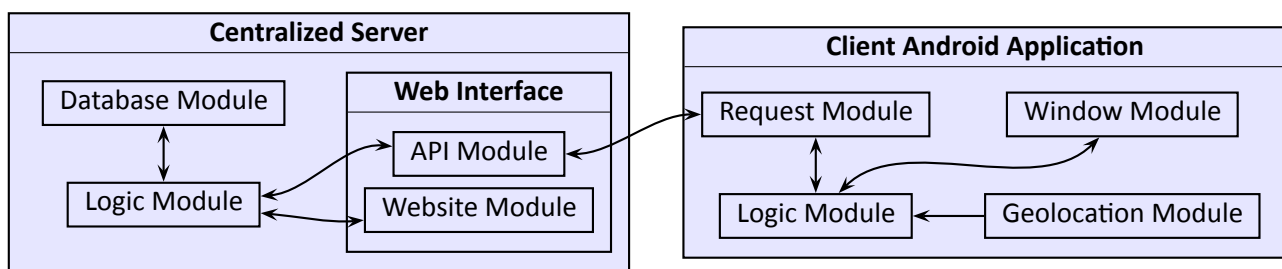


Figure 3.1: Data flow of the top level of the system, showing the application and server as clearly separated entities with internal modules that encapsulate distinct functionalities of the system.

As Figure 3.1 shows, the server and client have a similar overall structure. Both have a main contained logic processing module, a modifiable data source (the *Database Module* for the server and the *Request Module* for the client), and an interface (the *API Module* and *Website Module* for the server, and *Window Module* for the client). This compartmentalisation of processes is intended to make both the server and client subsystems more expandable and maintainable while features are being implemented and testing performed. The interaction between modules is restricted to a small and manageable set of interfaces to help reduce the internal complexity of the system.

3.1 Intermodule Dependencies

3.1.1 Database Module

This module will be used extensively by components in the *Logic Module* contained within the server, so considerable effort has been invested in designing a clean and powerful interface. The module will abstract away interaction with the DBMS (Database Management System), making it possible to easily replace the DBMS used to support different platforms. For the Windows operating system the server will be running on top of the .NET CLR (Common Language Runtime), and will therefore have access to Microsoft's SQL Compact Edition DBMS. On Unix systems, the server will be using the Mono CLR implementation, and

will therefore have to use an alternative DBMS, SQLite. As well as supporting different DBMS connections and SQL dialects depending on the host system, the Database Module will also provide a simple interface to common SQL operations through the use of code generation. This will reduce the amount of errors produced through the use of poorly constructed SQL statements by delegating the validation task to the debugging facilities provided by the IDE (Integrated Development Environment) used while developing the server.

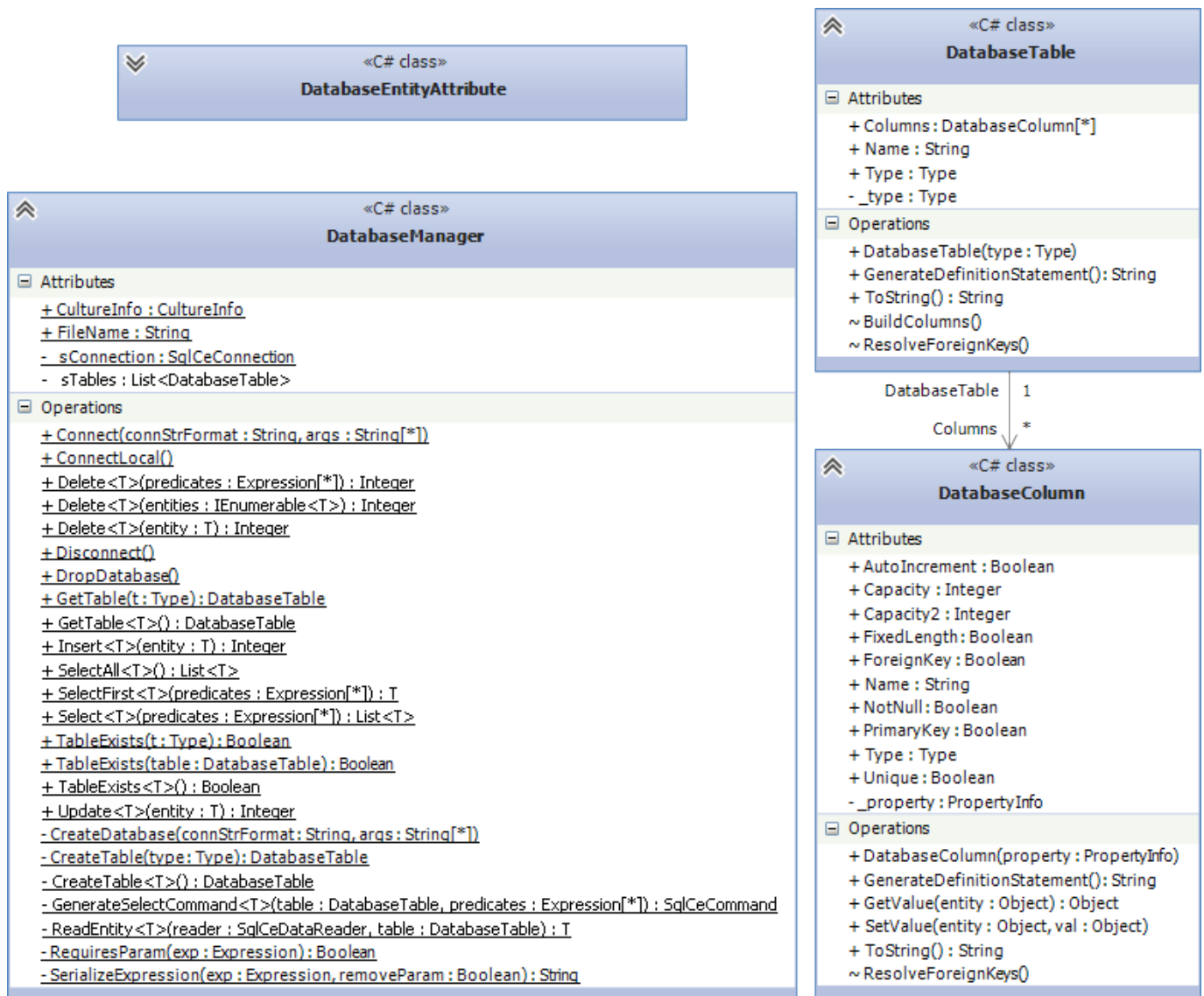


Figure 3.2: Class diagram for the main Database Module classes.

The three most important classes in the design are *DatabaseTable*, *DatabaseColumn* and *DatabaseManager*. *DatabaseTable* and *DatabaseColumn* relate to the tables and columns respectively in the SqlCe (or Sqlite) database. This abstraction of the database structure is used internally in the database module for the generation of SQL statements and also when reading information supplied by the database management system. Database table is simply a named collection of columns, which can be automatically be constructed from a C# class which has a *DatabaseEntity* attribute. When the server program initializes, all classes in the assembly are checked for this attribute. For each class that has one, a *DatabaseTable* is constructed and stored for later use in the *DatabaseManager*. If this table is not present in the actual

database, a DDL statement is automatically generated to define and construct it. This application of assembly reflection and code generation greatly reduces the amount of work needed to implement and use new database entity types. All that is required is the definition of a C# class with the *DatabaseEntity* attribute, and the fields to be stored marked with their respective attributes like the *PrimaryKeyAttribute* and *NotNullAttribute*. The class can then be immediately used in the code without manually adding it to any lists or writing a single line of SQL, and it will be stored in the database through a single call of *DatabaseManager.Insert()*.

The *DatabaseManager* class provides the bridge between the external DBMS and the internal representation of the database. It provides simple but useful functions to add, retrieve, and update items from the database. The aim with the design was to have a powerful interface without the need to write a single line of SQL when making a request. Hard coding SQL into the program would lead to a great deal more work when trying to support the two dialects used by the different management systems for each platform, and would require manual debugging instead of through the IDE's syntax validation. The query system will be implemented through the use of a translator that compiles C# LINQ (Language Integrated Query) expressions to SQL. Although an implementation of this functionality exists in the Microsoft implementation of .NET, the Mono one has no such feature. Therefore, the mechanism will have to be replicated specifically for this application.

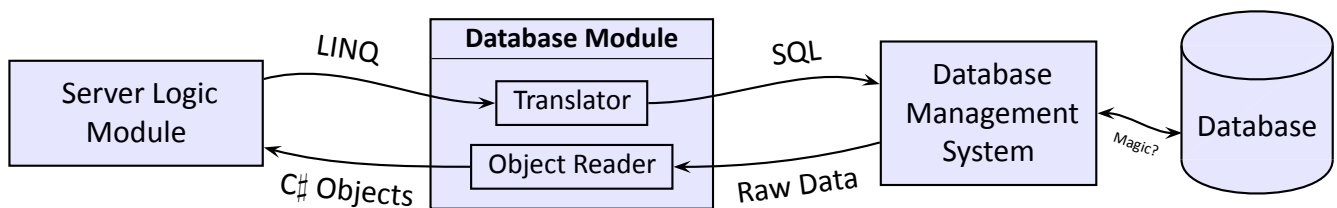


Figure 3.3: Diagram showing the data flow of the Database Module during a *SELECT* statement.

3.1.2 Logic Module

The Logic Module will contain all critical algorithms related to game and overall system state. This will include the operations of account authentication, client location validation, cache attacking, and unit transactions. The module contains a set of core classes that abstract components of the game such as caches and players. Keeping the more intensive calculations in the system separated from the database and interface means they are easier to locate and profile, and unit tests can be produced more simply without having to unnecessarily incorporate unrelated components.

This module is centred around processing the main elements of the game; players, caches, and their interactions. The *Account* and *Player* classes, which are *DatabaseEntity* types, represent the account details and game states respectively of a user. The separation of user data between the two classes is designed to match the conceptual difference, and also because users who have not been verified by email do not need *Player* objects since they are unable to participate in the game. The *Account* class will provide a wide array of static helper methods; to register a new user, validate an existing one, or promote a user to administrator status for example. The *Cache* class is also a *DatabaseEntity*, and represents a single cache. It provides a helper method *FindNearby* that is used both to detect whether a player is close enough to scout or attack a cache, and also whether a cache is too close to where a user wishes to place a new one. The other notable method is *Cache.Attack()* where the battle calculation is performed, deciding whether a cache is defeated by the attacking army or if the defenders keep possession of the stronghold.

Authentication sessions and notifications are also managed in this module. The decision has been made to not store authentication sessions in the database, but rather in memory. Authentication sessions are

quite volatile, in that they will be created and removed too frequently to warrant their storage in a sluggish but persistent structure like a database. They will be instead held in a dictionary structure, using the related account ID as a key for easy retrieval. Notifications are messages to users about a change in game state that may be of interest to them, or for messages sent to them by other users. These objects *are* stored in the database, and so the *Notification* class has the *DatabaseEntity* attribute.

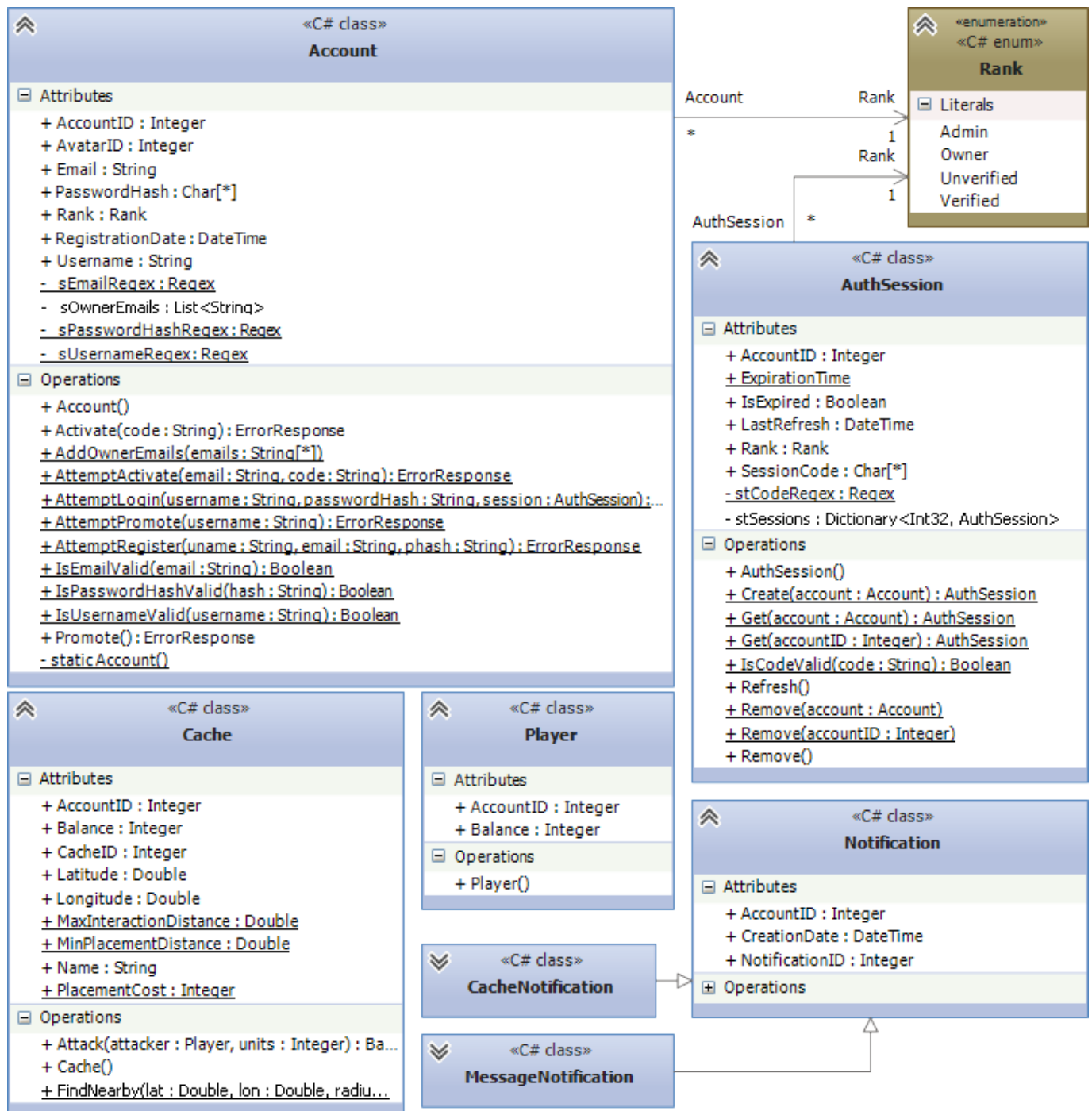


Figure 3.4: Class diagram for the main server Logic Module classes.

3.1.3 Web Interface

This module group is the interface in which incoming HTTP requests are processed and responded to. The requests come in two main categories; website resource requests and API requests. The website resource requests are from users browsing the website and requesting pages or static content such as images. These requests are directed towards the *Website Module*, which processes them and responds with the requested content. API requests are sent by the client application, and will be routed to the *API Module*. These requests are in the form of a command with a series of named parameters, and the response will be sent as a JSON (JavaScript Object Notation) object.

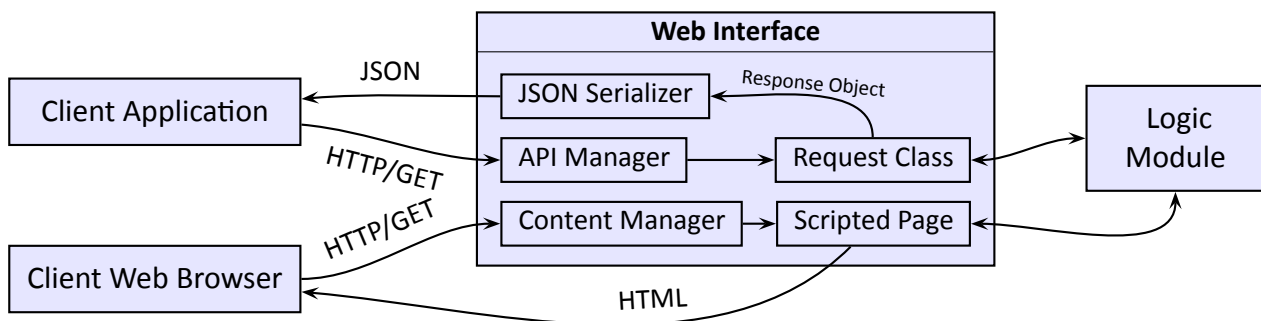


Figure 3.5: Diagram showing the data flow of the *Web Interface* module group.

3.1.4 Website Module

This submodule of the *Web Interface* handles the delivery of web content such as HTML pages, style sheets, and images. The content is served from a resource directory local to the server program, which is updated during the program's runtime. It also allows dynamic web page generation through the use of inline C# scripts in the source HTML files. These scripts are used as a preprocessing language while constructing the page, so the resulting file sent to a user's web browser has content reflecting the current state of the game and the actions of the user. The inline preprocessor scripting system will be implemented specifically for this project, using code generation and .NET's "compiler as a service" facility. The choice to do this instead of simply using PHP or any other pre-existing CGI is so the scripts would have direct access to the server program and its components (this would be true because the scripts are actually contained within the server program itself). This would provide increased performance and would eliminate the need to produce an interface between the CGI and the server.

3.1.5 API Module

The second submodule of the *Web Interface* group handles the interpretation and responses to requests from the client application. These requests arrive in the form of a specific command, and the named parameters required by that command. The module contains a class for each command, all extending a general *Request* superclass that provides some facilities common to most requests such as authenticating the requesting user. After the request is processed, an object extending the *Response* superclass is returned by the processing request class. This object is then serialized procedurally into a JSON object, which is sent as a reply to the requesting client. The separation of request and response types into classes has the standard benefits of reduced code dependencies and therefore complexity, and also allows requests (or responses) with similar functionalities to extend general abstract classes that implement them. JSON is

used as a response format due to its small bandwidth footprint (satisfying requirement 5.2.30) and how trivial it is to parse. It also opens up the possibility of using it with AJAX for the website implementation.

Assembly reflection will be used to automatically detect the corresponding class for each request, and then create an instance of it. All *Request* classes will override a *Respond* method with a *Response* return type, which will be invoked upon instance creation.

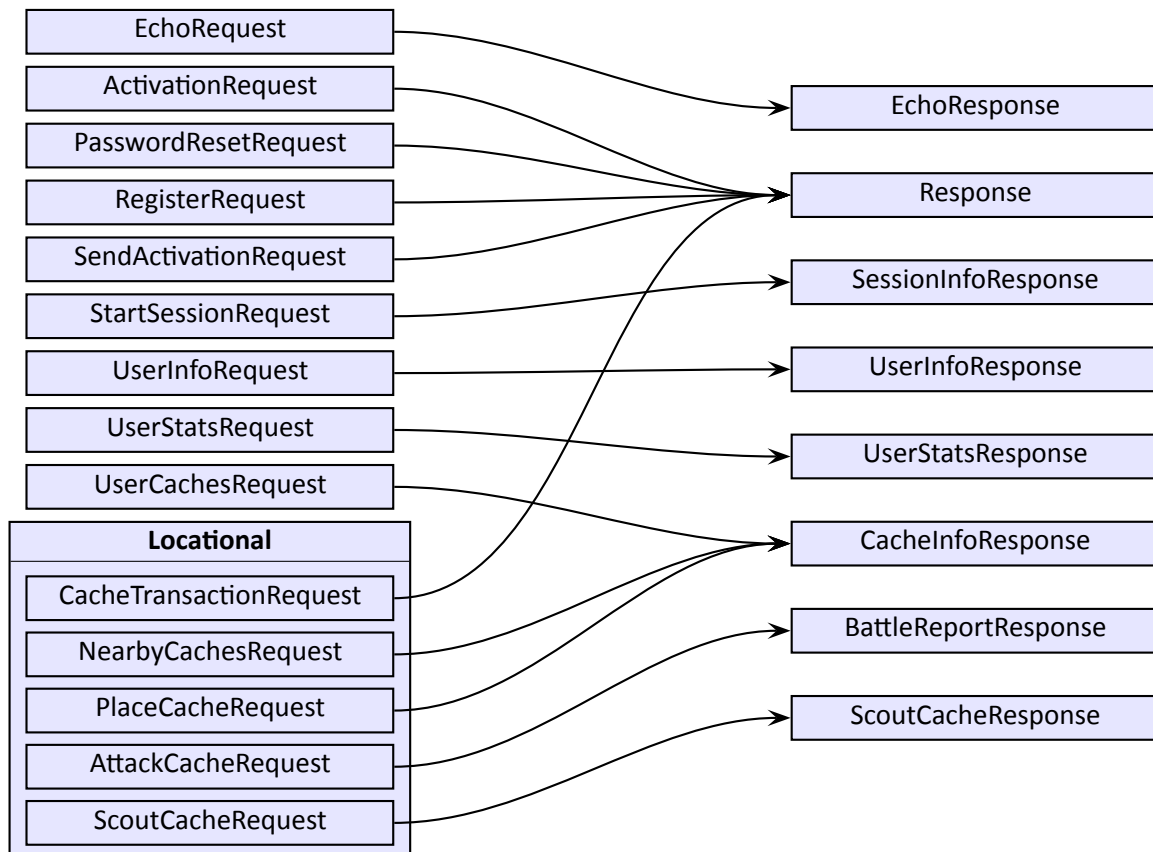


Figure 3.6: A list of each *Request* and *Response* type, and the relationships between them.

The *Response* superclass has one mandatory attribute - a boolean marking whether the original request was successful or not. For all but one of the extending subclasses, this flag is set to true by default. The one class that doesn't is the *ErrorResponse* class. This class also includes a *String* attribute containing a message describing why the request failed. The mandatory inclusion of the *success* boolean means the receiver of the response can tell if the response object will contain the desired output, or if there will be an error message to read. Standardising the error format also allows response parsing on the client end to be simplified and nicely abstracted too.

3.1.6 Request Module

This client application module handles the construction and sending of requests to the central server, and then parses and processes the responses given. Each command has a distinct class encapsulating the parameters required and the action to perform with the response. The request classes that have parameters in common extend superclasses implementing those parameters to avoid information duplication and therefore improve maintainability. Other components of the application will, when required, construct a request object instance of the desired type, populate the needed parameters and reference an event handler class to invoke on response arrival, and initiate the request. The request is carried out asynchronously.

A flag in the request object will be toggled when a response arrives, and the given event handler triggered. This asynchronous event based pattern is designed to work well with a graphical user interface that must not block execution while waiting for a request, and also allows multiple requests to be performed at the same time.

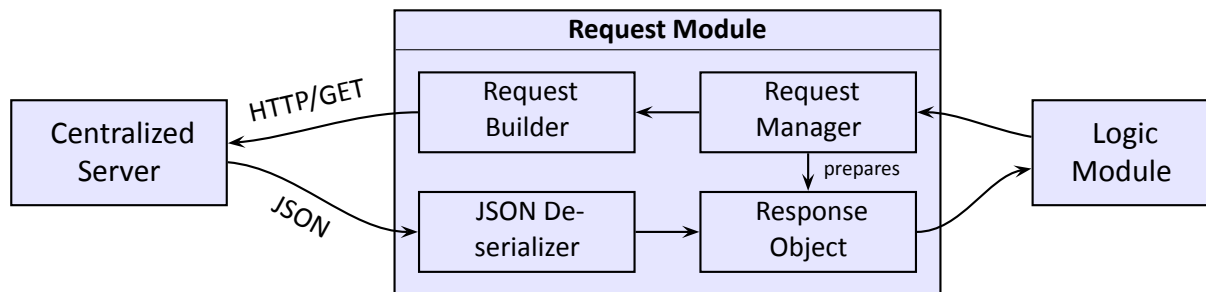


Figure 3.7: Diagram showing the data flow of the client application *Request Module*.

The *Request* and *Response* classes in this module will mirror those of the server, with the main difference being that the *Request* is the one to be serialized (into a HTTP GET request), and the *Response* is deserialized from the JSON sent by the server.

3.1.7 Logic Module

The logic module is where the major processing and validation algorithms of the client application reside. It holds an abstracted model of the game state relevant to the user of the client application, such as data about caches within range of the host device, and profile information about the current user. This module decides what actions the user can perform at each instance in time, and whether an action performed is valid. It also decides what information should be shown to the player, but leaves the details of how it is displayed to the *Window Module*. The components of this module are abstracted for the same reasons as the logic module in the server, to make them easier to locate, unit test, and profile.

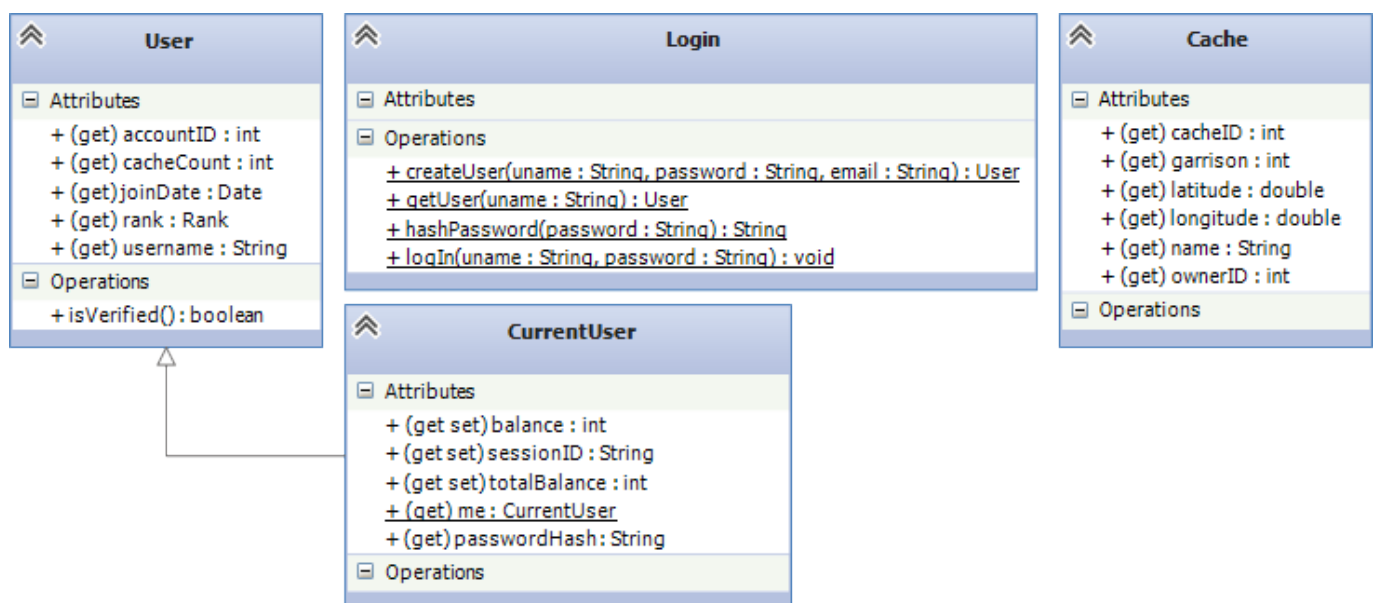


Figure 3.8: Class diagram for the main client Logic Module classes.

3.1.8 Geolocation Module

This module has two main purposes, to encapsulate the process of finding the current GPS position of the host device, and to interface with the Google APIs which provide various location based services. The module abstracts the methods used for both functions so as to allow for easy refactoring of the internal implementation, reduce intermodule dependencies, and therefore reduce code complexity. The separation of this set of components from the rest of the system should also make the process of porting the application to other systems such as iPhones easier, since the action of reading from GPS sensors is quite low level and operating system specific and it would be difficult to find all instances of the action if it was spread all over the project.

3.1.9 Window Module

The window module will be the largest of the application modules. It encapsulates the graphical user interface, including the presentation of data to the user and the interpretation of user input. The user interface is divided into "screens", which are individual views that may be displayed to the client. All screens will inherit from a main abstract superclass that provides the skeleton functionality required by each view. Cleanly separating the application logic from the user interface has numerous benefits; it promotes code decoupling, improves maintainability and the ability to test and profile the code, reduces the amount of information duplication where algorithms are copied for several different views, and means interface layouts and styles can be found and edited with greater ease.

Each screen will contain components such as buttons and text entry fields. User input through these fields is initially validated where appropriate by the screen, but major functionality is delegated to the *Logic Module*.

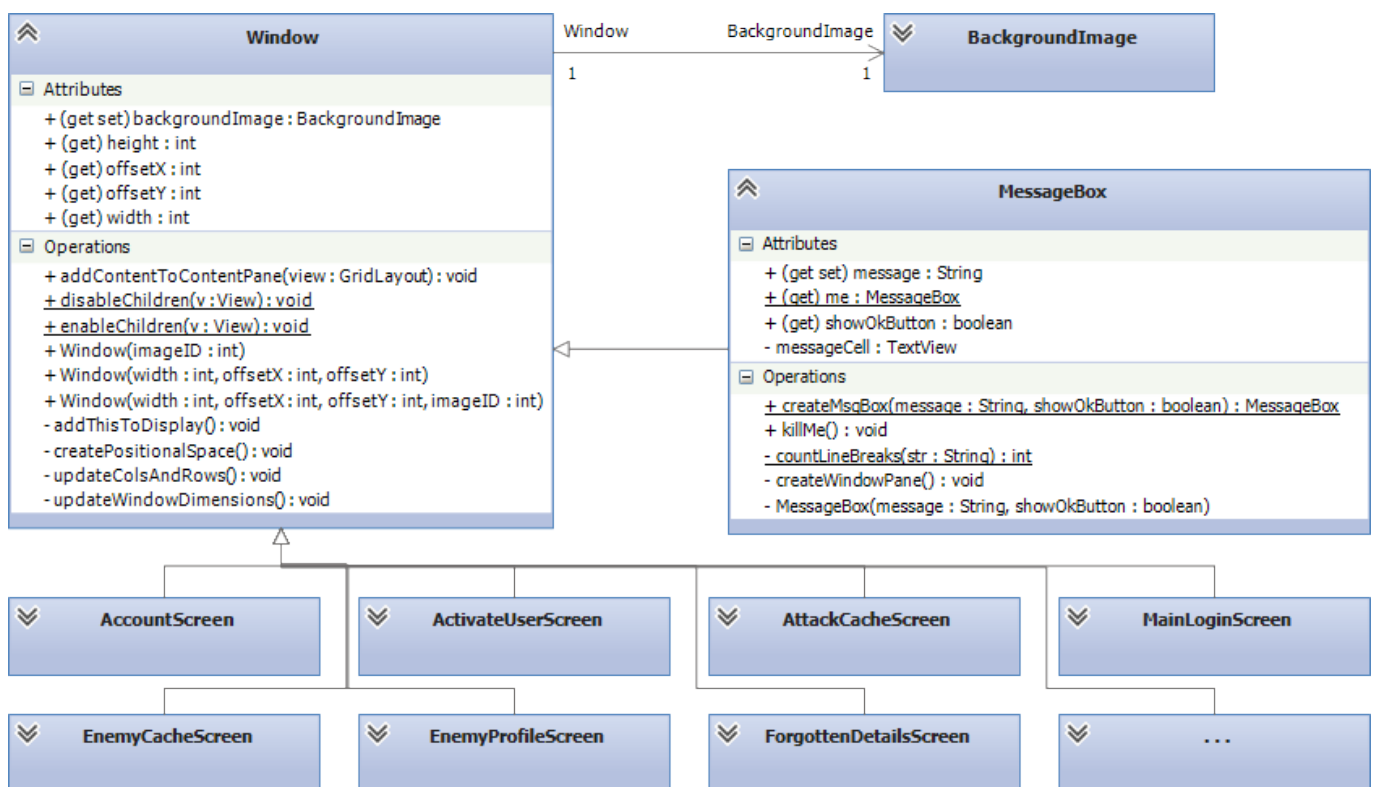


Figure 3.9: Class diagram for the main Window Module classes and a selection of the *Screen* classes.

3.2 Interprocess Dependencies

This section will cover the design of the dynamic behaviour of the system through description, sequence diagrams and activity diagrams. It will start by looking at the android application.

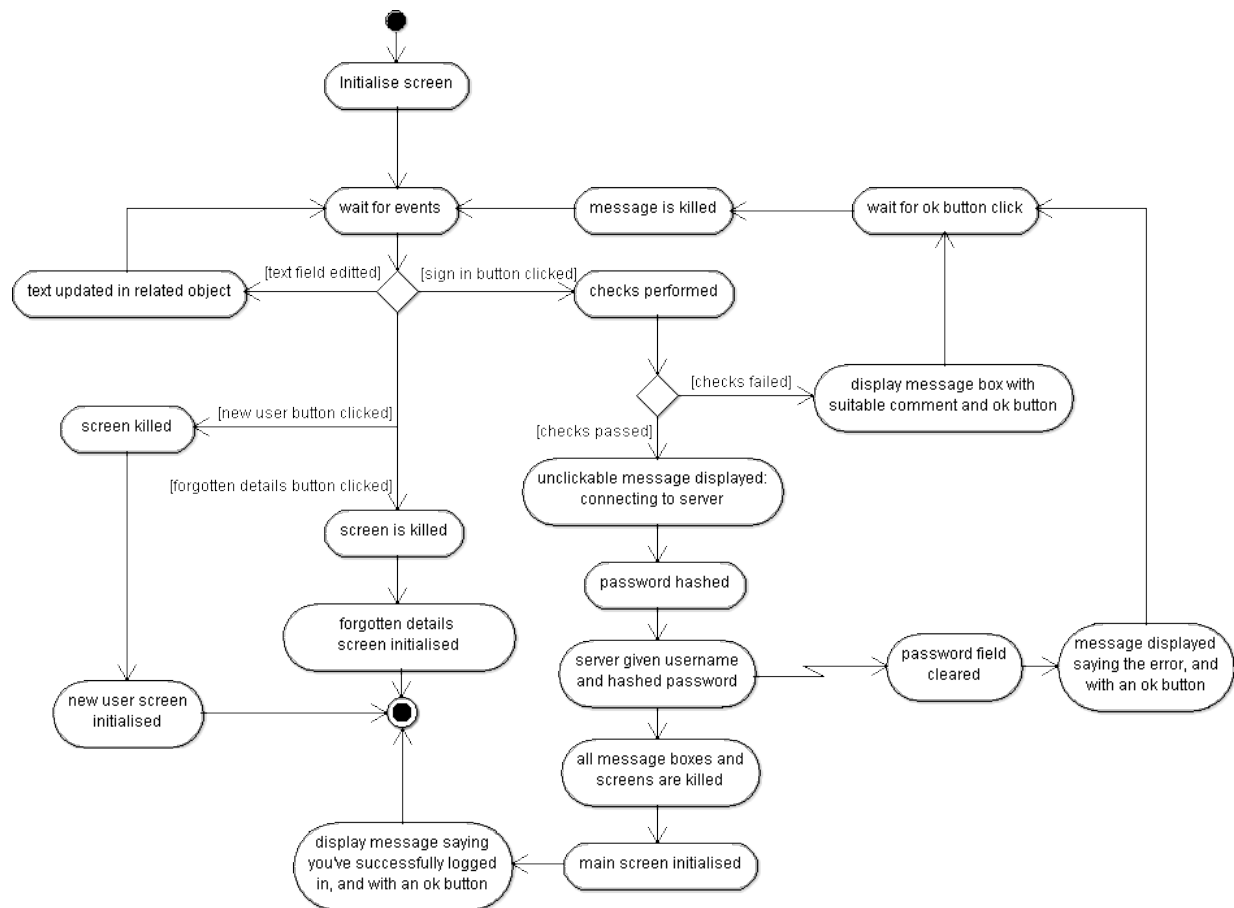


Figure 3.10: Activity diagram for the main login screen.

When a user first starts the app they will be taken to the main login screen. Figure 3.10 shows what can happen from this screen. Firstly the screen is initialised and graphics, buttons and text fields are put in the grid that makes up the view. After this the app will wait for events from the user, these include typing into the text fields and clicking buttons. If text is typed it is saved in the relevant text field and if the new user button or forgotten details button are clicked then the main login screen is killed and the relevant screen is initialised. However if the sign in button is clicked, the following processes are a bit more complex. Firstly the text fields are checked and if they are null a message is displayed. This message box will have an OK button that once clicked will kill the message box and the app will wait for more events. If the text fields aren't empty then the password is hashed and the hash and username is given to the server. If there are any errors connecting to the server or creating the user session then a message box will be displayed with the error and an OK button. If not the user will be logged on and shown the main screen and a message box saying successfully logged on, with an OK button. This satisfies requirement 5.1.5 user authentication.

Similar processes will occur for creating a new account from the new user screen, except the checks performed include making sure text fields aren't null, the username and password are 5-16 characters long, the password and check password fields contain the same String as the email and confirm email fields, the password has a letter and digit and the email is valid. In this case the server will be given the username

and email as well as the hashed password and if there are no errors with the server the user will be shown the main login screen with a message telling them that an email has been sent to them to activate their new account. This satisfies requirement 5.1.1 account registration.

There are also few differences when a user has forgotten their password and has clicked the forgotten password button. The email address text field must be checked to see if a valid email has been entered. If the server request is successful then the user will be shown the main login screen with a message telling them that an email has been sent to them with a link to refresh their password. This satisfies requirement 5.1.8 password reset.

Sequence Diagram

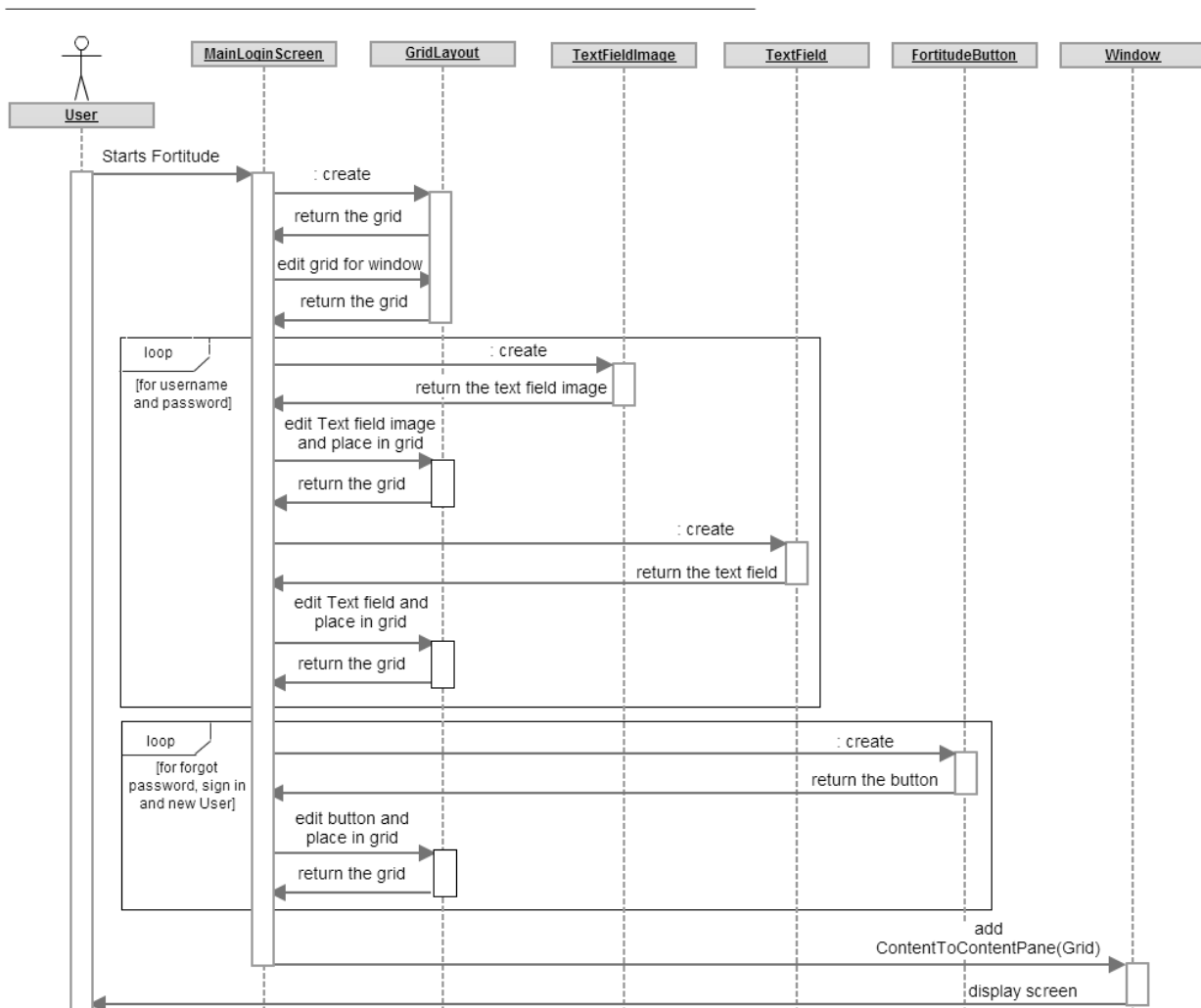


Figure 3.11: Sequence diagram for the construction of the main login screen.

Figure 3.11 shows what happens in each of the classes when the main login screen is initialised. It is very similar for all screens. Once a new MainLoginScreen is created, the grid which everything sits on is initialised in the GridLayout class. The MainLoginScreen transforms the grid so it has enough squares and is the right size. Instances of TextFieldImage and TextField are initialised, edited and placed in the grid for username and password. Then the buttons for forgot password, sign in and new User are initialized in FortitudeButton, edited in the MainLoginScreen and placed in the grid. This grid is put in the window using a method in the Window class and this method displays the window to the user. When creating the grid and everything that goes in it the sizes are decided on the window size so nothing will be off the screen,

satisfying requirement 5.3.8.

Figure 3.12 shows the sequence of events when the sign in button on the main login screen is pressed. Firstly the MainLoginScreen will call the method `signIn()`. It checks to see if the text fields aren't empty and displays a message box if they are. If not a message box saying connecting to server will appear which should satisfy requirement 5.3.9 as the rest may take a while to be performed but the user will see the message box nearly instantly. Next `initialLogin()` is called in the Login class which will hash the password and give the username and hashed password to ServerRequests. This will create an object of type RequestThread and call `setURL()` to give it a URL with all relevant information. This object connects to the server using the JSON parsing module and will analyse the response. JSON is used because it is very 'light weight' to satisfy requirement 5.2.30. In the diagram I've just talked about a successful response. However if an error occurs the response will show this so ServerRequests can create a MessageBox with different text and not create a CurrentUser or display the main screen. But if the response is a success then these things will happen and a message with an OK button will be displayed that tells the user they've successfully logged on. This class diagram is very similar to other processes in the app as values are always checked in the current screen and the ServerRequests and RequestThread class as well as the JSON parsing module are used for most actions with a different method call in ServerRequests, therefore I will not include further class diagrams exactly how methods in ServerRequests work. Values are also checked by the server, however they are initially checked by the application as it will not take much processing time so the user will get a faster response if they've made a mistake.

Sequence Diagram

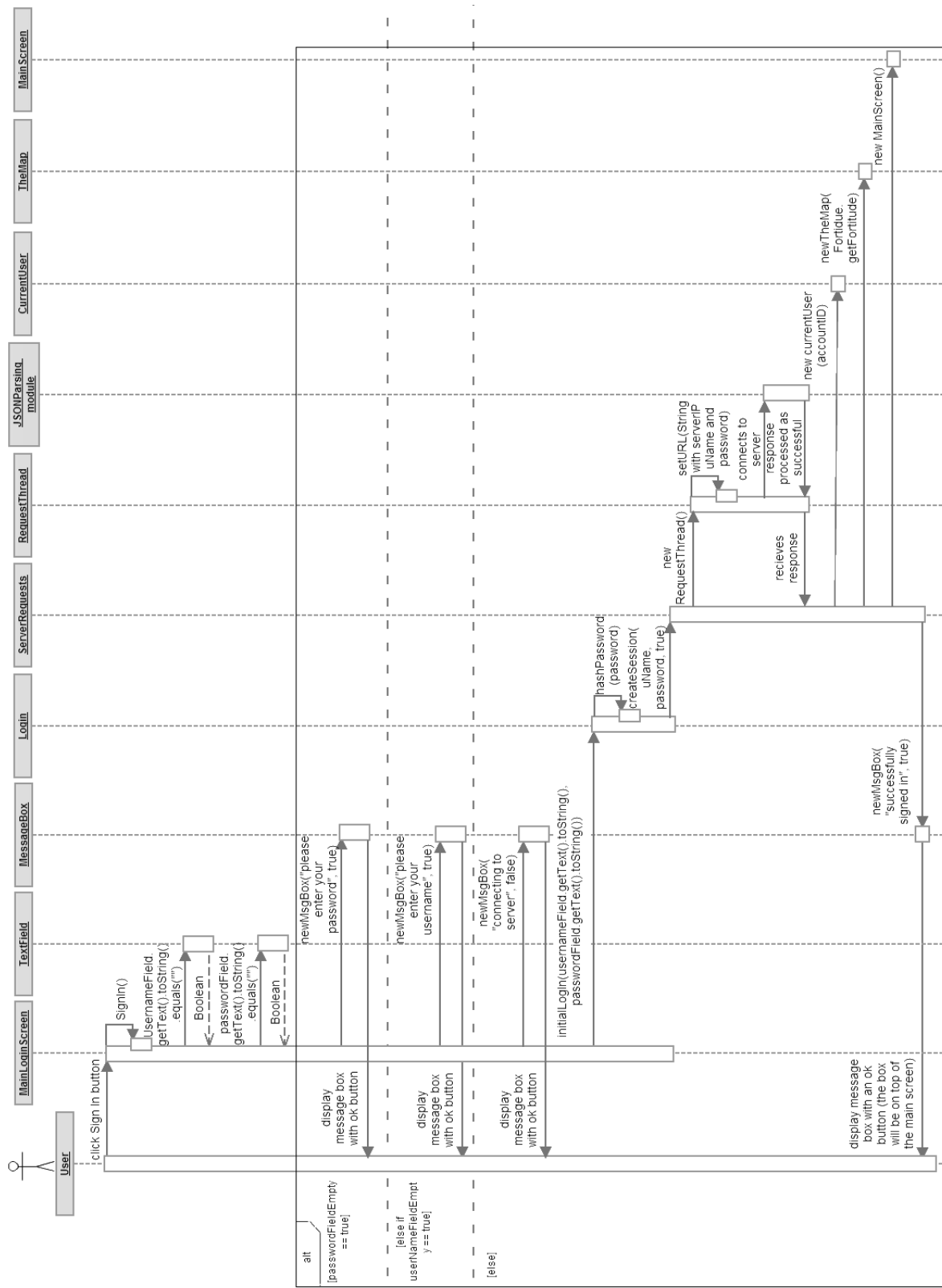


Figure 3.12: Sequence diagram for the sign in button.

Before the main screen is shown the map must be created as it is viewed on the main screen. If it has already been created it is returned else a new map is created. When this happens a map view is created and added to the map and particular Google map settings are enabled, including zooming to satisfy requirement 5.3.3. Then threading is used to run two parallel processes, one gets the users position and set's the map to be showing this position, (satisfying requirement 5.3.1) get's nearby caches and set's them as markers on the map with different colours and types relating to owners and cache types, to satisfy requirement 5.2.26. This has a separate method so that caches can be loaded around the user's position when requested to satisfy requirement 5.3.2. When getting the nearby non-player caches, it will only put them on the map if the user hasn't attacked it for a certain amount of time, satisfying requirement 5.2.19. The other parallel process deals with user interactions, putting caches on the map after the user refreshes it and listens for when the user clicks a cache to bring up a new related screen.

If a user has logged on before they will be automatically signed into that account again as shown in figure 3.14. After the MainLoginScreen has been initialised it will create an instance of FileSave and use it to get username and password hash in String form the byte form in files that the app stores. If the username and password aren't null or empty then the ServerRequests class can create a session for the user from them, and take the user to the main screen. However if the username and password are empty or null then nothing will happen and the user will not be told they are connecting to the server but shown the main login screen. When a user logs out through the AccountScreen, their username and password are put as the empty string using the createDialog() method in FileSave.

If a user's account is not already activated then the bottom bar in the main screen will not have the normal buttons but instead have a button to take the user to the ActivateUserScreen. From this screen the user can click a button to resend the activation email which uses resendActivationEmail() in ServerRequests with the email of the account. This satisfies requirement 5.1.4.

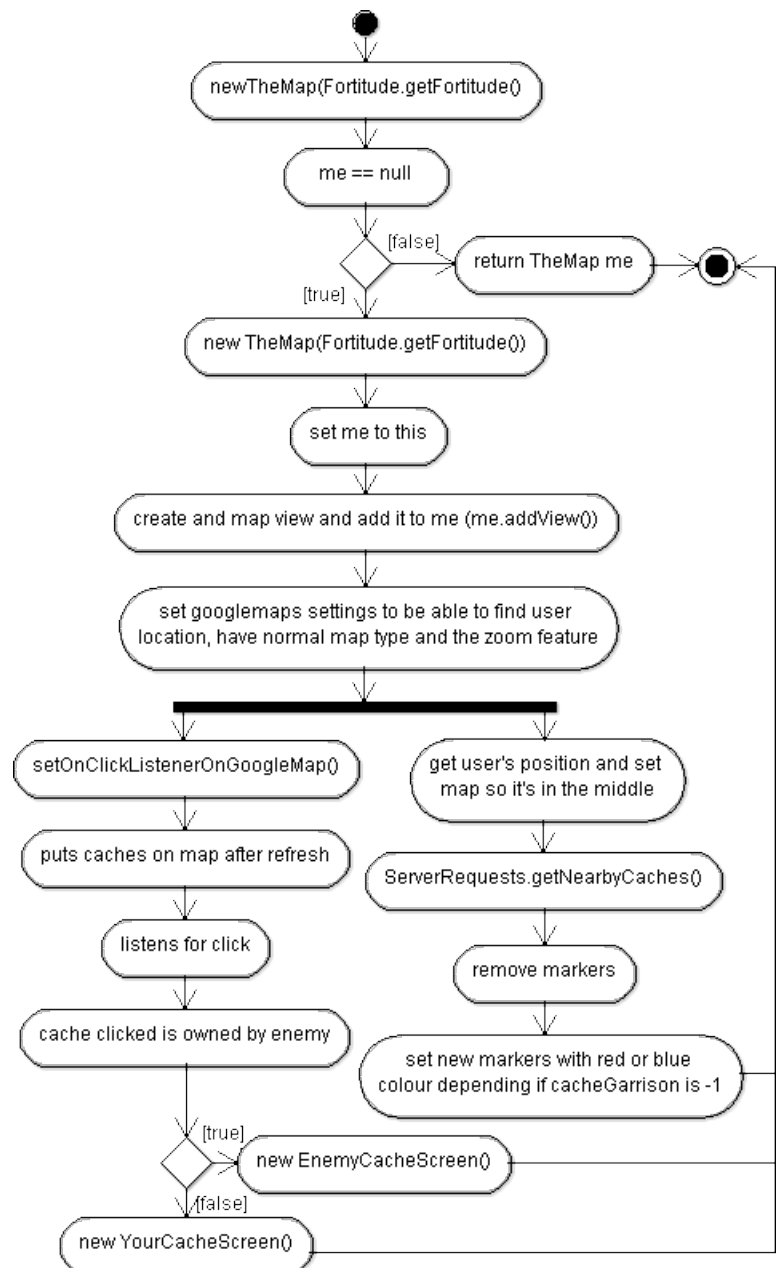


Figure 3.13: Activity diagram for construction of the map view.

Sequence Diagram

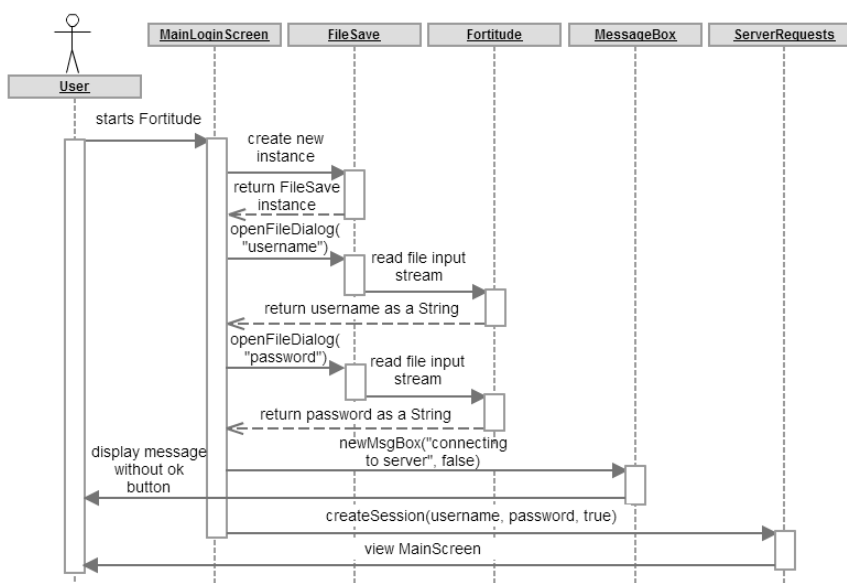


Figure 3.14: Sequence diagram for the credential autosave feature.

The sequence diagram for clicking the castle button is more complicated than for clicking other buttons so is shown in figure 3.15. For these events to occur the castle button must be clickable. If the castle is clickable a message is shown telling the user they're being connected to the server. Then `refreshData()` is called in the `ServerRequests` class to make sure all information including cache positions and user data is up to date. `refreshData()` can update a route to a cache too if there is one. If this request isn't successful then a message with an OK button is displayed to the user that says loading caches failed. If the request is successful then the latitudes and longitudes are found out

for the user's current position and caches on the map and nearest cache and distance to it is calculated. If the closest cache isn't close enough then user is shown a message box with an OK button that tells them to get closer to the cache. Otherwise if the cache is owned by them they go to the visit your cache screen, which will show the cache balance satisfying requirement 5.2.4. However, if the cache isn't owned by them they go to the visit enemy cache screen which will say the owner of the cache. If it's a non-player cache the visit enemy cache screen is still used, however it will not have the owner of the cache on. This will satisfy requirement 5.2.3, cache ownership.

Sequence Diagram

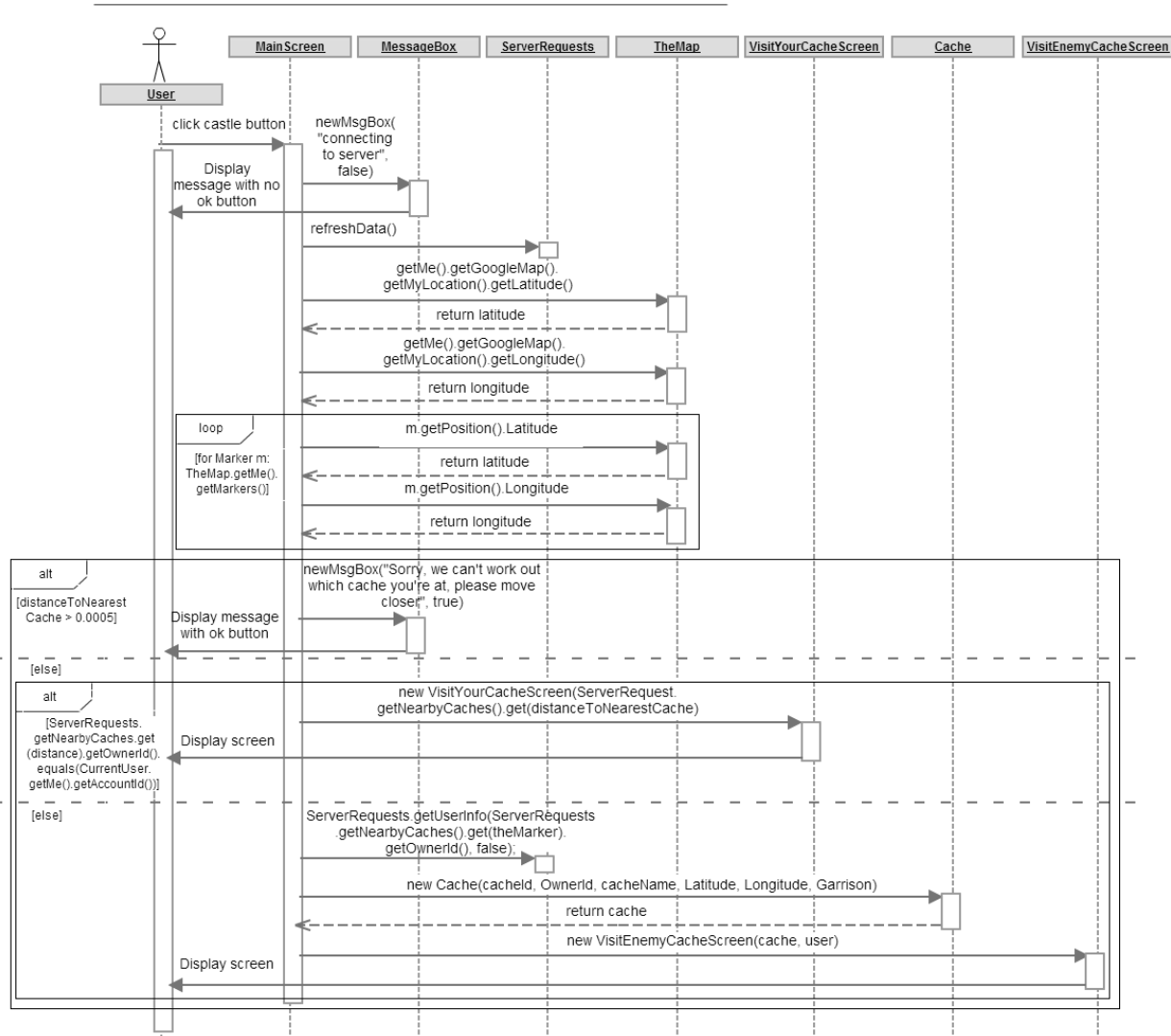


Figure 3.15: Sequence diagram for the context sensitive castle button.

As previously said the IconUpdater class decides whether the castle button is clickable and also what image is displayed for it. Once created, it runs in the background of the app unless it's told not to. While it hasn't been told to stop running it tries to get the position of the user and the positions of caches. If any cache is close enough then if the main screen is displayed the castle icon is viewed as not greyed out and made clickable. Else if no cache is close enough and the main screen is displayed the icon is viewed as greyed out and changed so when clicked nothing happens. If IconUpdater gets an error doing any of this more than 10 times the user is told that the IconUpdater has crashed through a message box.

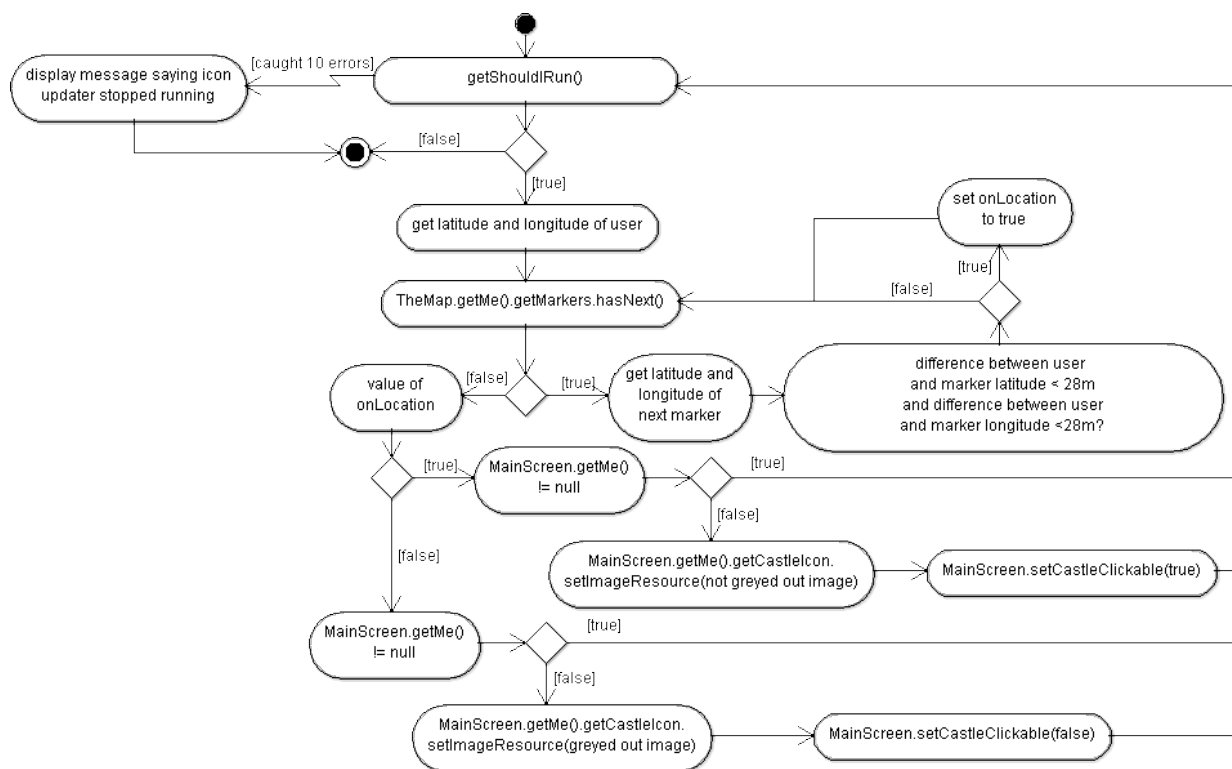


Figure 3.16: Activity diagram for the castle button icon updater.

The castle button is used to attack and scout caches. When the scout cache button is clicked the app checks it can access the phone's location and Google maps. If it can't it will tell the user the problem else the user is told they are being connected to the server. The user sends a request to the server to scout a cache using the phone's location, the users chosen number of scouts to send and the cache ID. If there is a problem then it is explained to the user. The server is then asked to refresh the data so any scouts that are lost are updated on the app. Again the user is told any problems. When the first request was sent a response was made and the app analyses this to find out which screen to display next. If all the scouts sent were lost the scout failure screen is displayed else the scout success screen is displayed. This satisfies requirement 5.2.9 cache scouting.

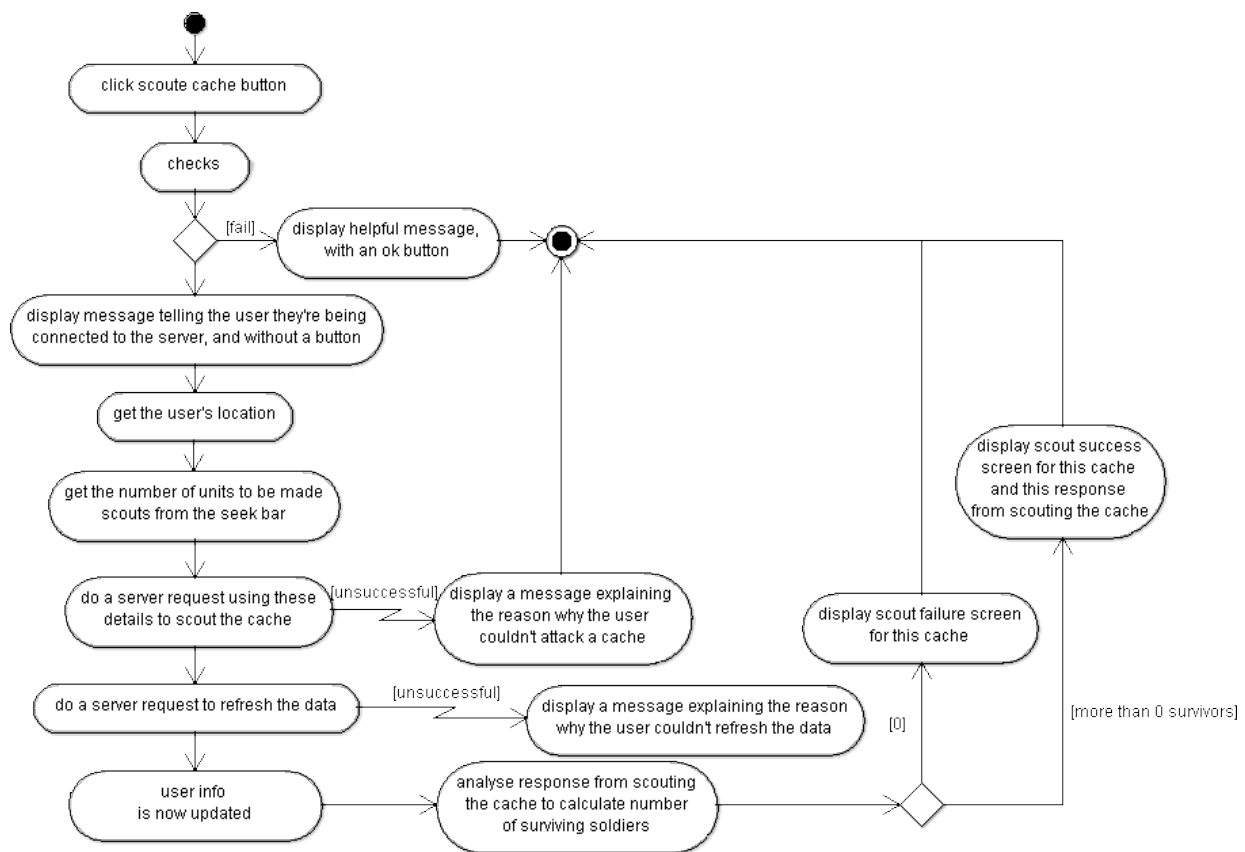


Figure 3.17: Activity diagram for the cache scouting interface.

A user can attack caches by clicking on an attack cache button. The app checks the location can be found and tells the user if their location can't be found. If it can be found the user is told they're being connected to the server. The user's location, number of units they wish to fight with and the ID of the cache are used to make a server request to attack a cache. The server is then requested to be refreshed so all the data can be changed based on the outcome of the attack such as the new cache balance from the remaining defenders or attackers and the owner of the cache. This satisfies requirements 5.2.11 successful attack and 5.2.12 successful defence. If there are any problems with these requests the user is told. If not the user is displayed the post battle screen which processes and displays the response from the first server request, which will satisfy requirement 5.2.13 battle breakdown. This satisfies 5.2.10 cache attacking. Attacking non-player caches is very similar, however the refreshing data might change whether or not the cache is viewed on the map if the user wins the battle, also the owner and cache balance will never change. This satisfies the requirements 5.2.17, 5.2.18, and 5.2.19.

Sequence Diagram

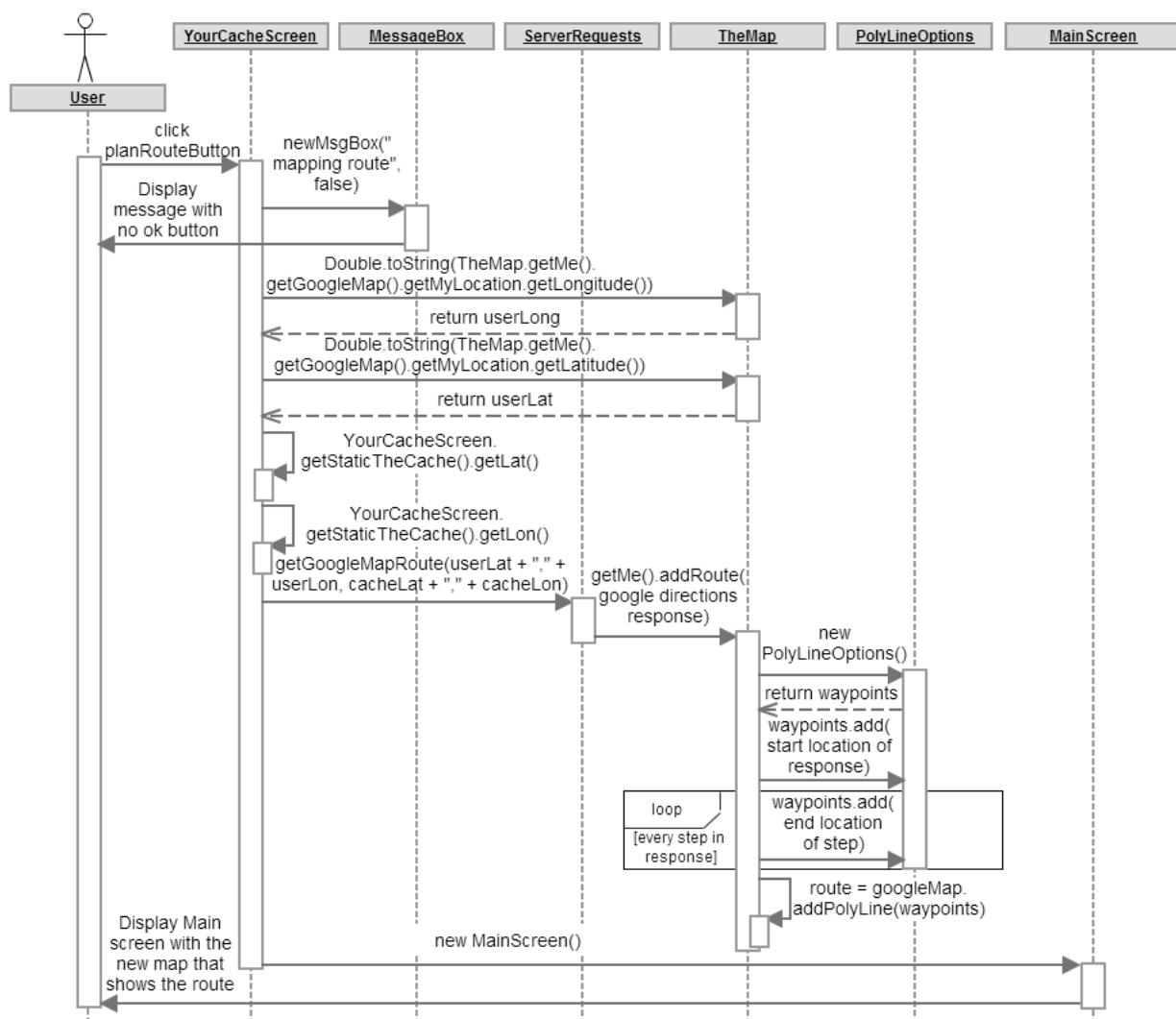


Figure 3.18: Sequence diagram for the route finding feature.

A user can get to caches using a plan route button. In figure 3.18 I've used the one on YourCacheScreen as an example but you can similarly plan routes to enemy caches. Firstly when the user clicks the button their location is checked and if it can't be found they are told. If not they are told the route is being mapped. TheMap class is used to find the users location and the class of the screen they're on is used to find the caches location. The locations are used as Strings for the getGoogleMapRoute() method in the ServerRequests class. This calls the addRoute() method in TheMap class with the response received. This method uses a predefined class to create waypoints out of the response and join them up on the map to make a route. Lastly the user is taken to the MainScreen with the route on the map. This satisfies requirement 5.3.4 path finding.

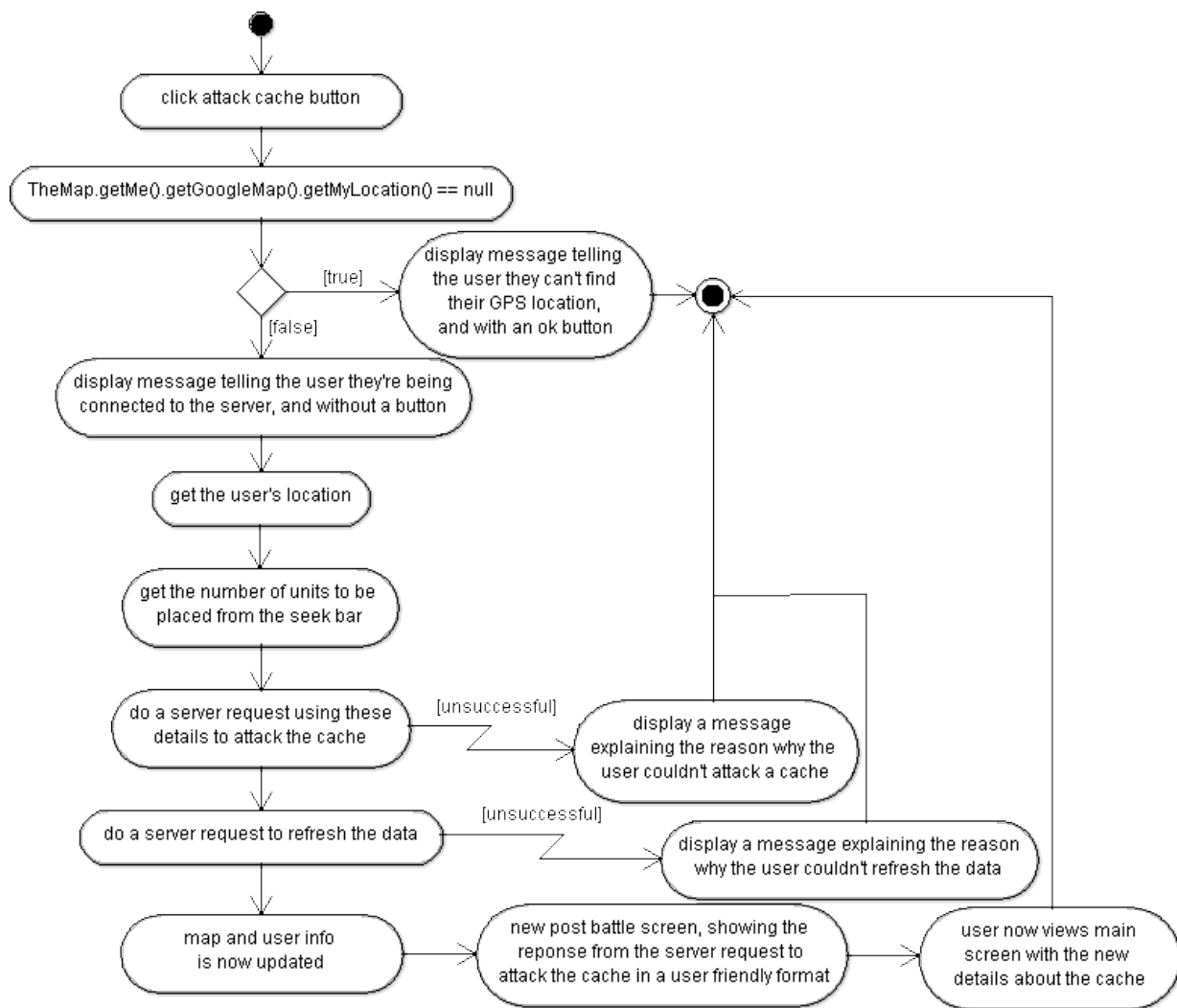


Figure 3.19: Activity diagram for the cache attacking interface.

A user can place a cache by clicking the flag button. If their location can't be found they are told, otherwise they are told they're being connected to the server. A server request is made using the user's location and number of defenders they wish to make defenders for the cache to be placed. If this is unsuccessful e.g. due to an error, the user trying to deposit more soldiers than they have in their army, or being too close to another cache then the user is told, else the server is requested to refresh the data for the new cache to be added. Again, if this is unsuccessful the user is notified why, else the user goes to a new main screen with the map now showing the cache. This satisfies requirement 5.3.5: Cache placement. It should be noted that when a cache is placed, there must be at least 5 units in the seek bar, 4 of which will be lost. This satisfies requirement 5.2.8: Cache placement cost.

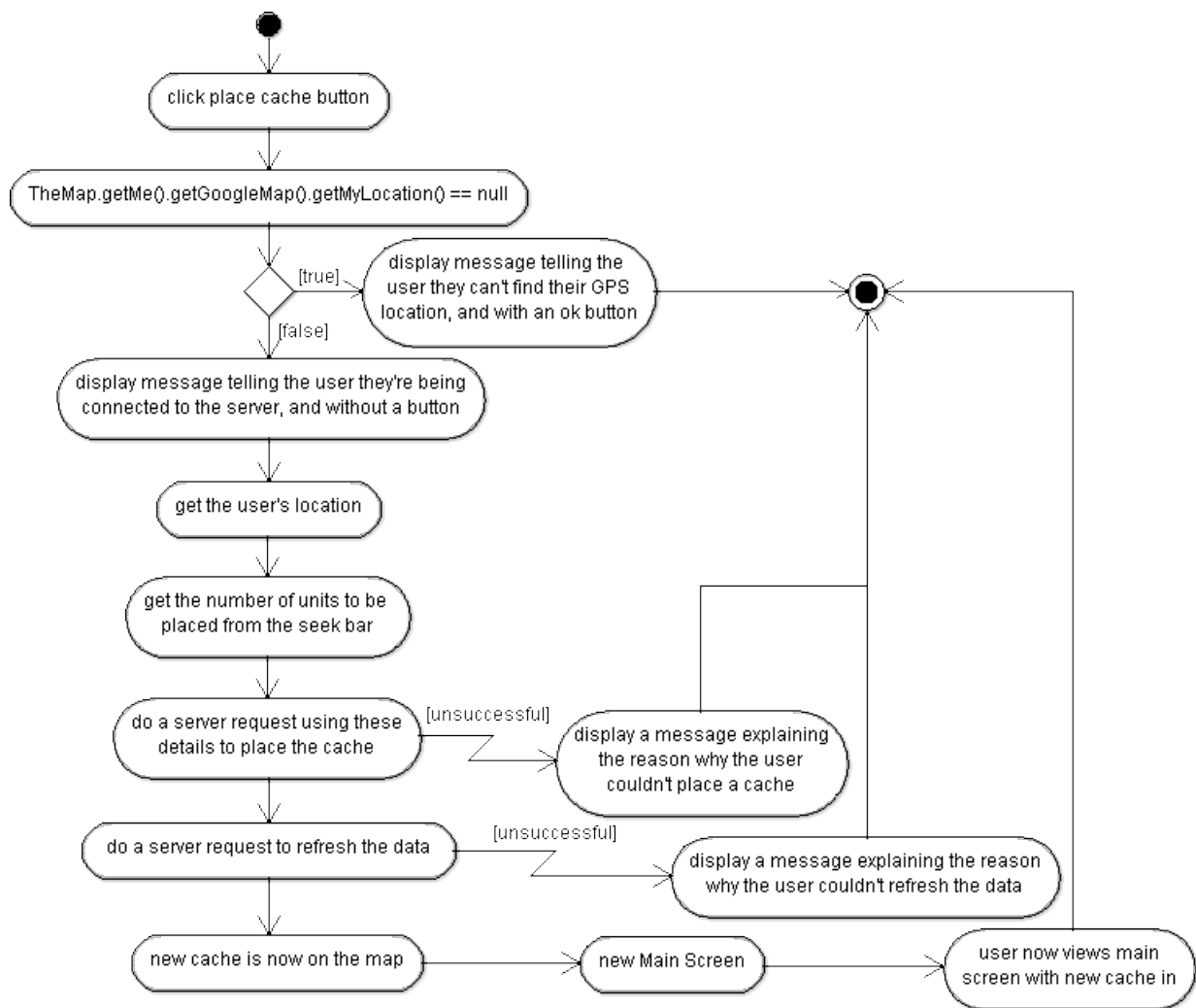


Figure 3.20: Activity diagram of the cache placement interface.

On the cache transactions screen the user can withdraw and deposit soldiers from the cache. When the submit button is pressed the users location will be found and if it's null then the user will be displayed an error message, if not they will be told they are being connected to the server. The number of units to be withdrawn or deposited will be found from the seek bar, `ServerRequest.cacheTransactions()` will be called then `ServerRequests.refreshData()`, to be able to view the changed balances. If any errors occur in the `ServerRequest` methods then the user will be explained the error in a message, otherwise they'll be taken back to the main screen. This will satisfy requirement 5.2.5: Cache transactions, and 5.2.7: Cache depositing. If the user withdraws all the troops then the cache will become unowned, satisfying requirement 5.2.6: Cache withdrawal. Unowned caches will have a separate screen from which users can place soldiers in to own that cache, further satisfying requirement 5.2.7.

There is a messaging feature in the app, which can be navigated around from the inbox screen. This screen will be made like most other screens except for the buttons to particular messages, but a few more classes are required before adding these to the grid. Firstly the information about the viewable messages are set using `ServerRequests.setMessages(username, when)`. The `when` variable is used to tell which messages are used, for example if the user clicks next twice then the 3rd set of messages will be used, then if the user clicks previous the 2nd set of messages will be used. There will be checks in place to stop 'when' being invalid for example negative or so large there are no messages in the set. The message information

Sequence Diagram

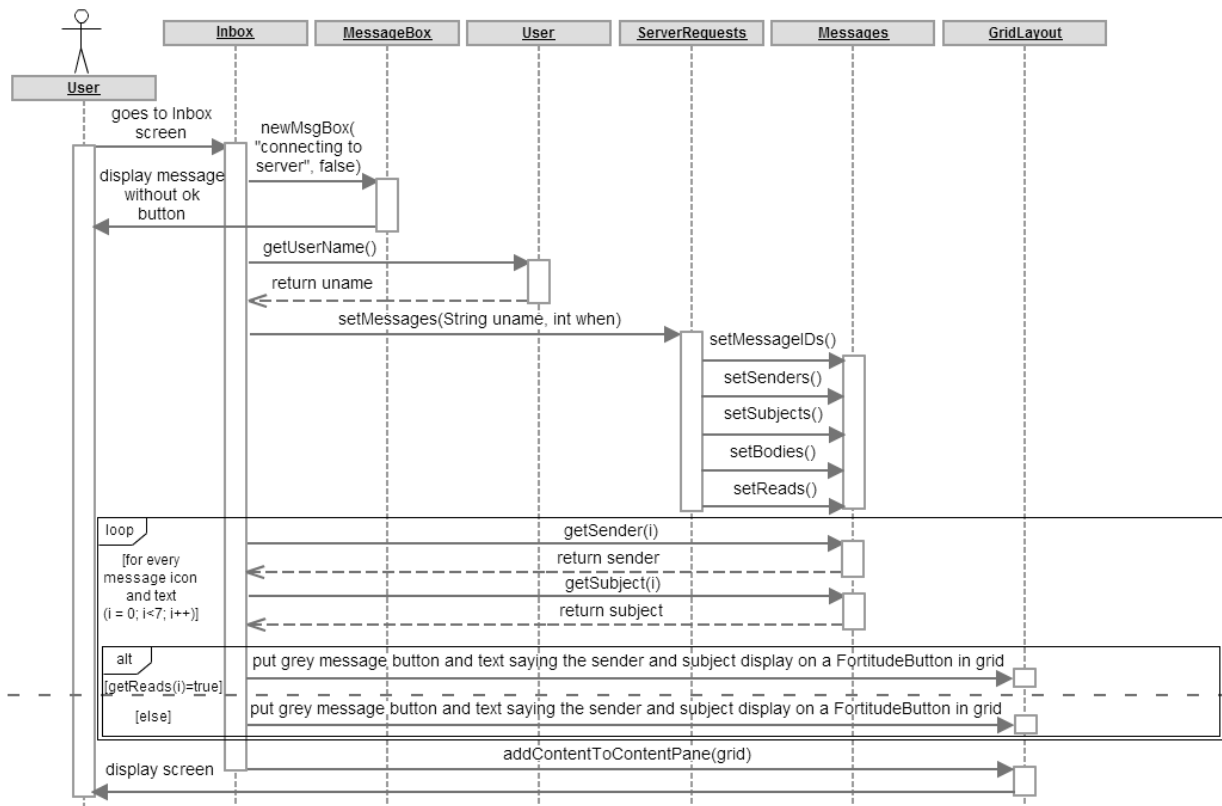


Figure 3.21: Sequence diagram of the messaging feature.

is used to decide what the icons look like and the text displayed. This satisfies requirement 5.4.1: User communication.

When a user clicks an icon to go to a message the variables related to currentMessage are set to those from the corresponding message in the array in the Messages class and if currentMessage.getRead() is false then ServerRequest.setRead(currentMessageID) is called and currentMessage.setRead(true). The values in currentMessage are then used to view the message screen, to automatically put the correct user in the 'send to' box when replying and to feed into the server request reportMessage if a user wishes to report the message. This satisfies requirement 5.4.2: Communication reporting.

Cache reporting works much like message reporting and is done through the same screen. When reporting a cache, the screen will have a subject and description relating to caches rather than messages, and when submit is pressed reportCache() in ServerRequests will be called with the cache, the user and the text that was entered as parameters. This satisfies requirement 5.2.22: Cache reporting.

Users can block other users through the settings menu in messages and through the EnemyProfile-Screen account screen. When a user is blocked, User.getBlockedUsers() is called to check if the user is already blocked and if not User.addBlockedUser(String username) is called then the method blockUser(String userBlocker, String userBlocked) is called in ServerRequests. The information is stored in the app and server as it will speed up loading screens to have it in the app while it has to be in the server so when other users messages aren't sent they can be notified. When a user is unblocked, the User class checks they are blocked and removes them from the list of blocked users using User.removeBlockedUser(String username), then the method unBlockUser(String userBlocker, String userBlocked) is called in ServerRequests. A user is blocked and unblocked using the same button in the EnemyProfileScreen so when this screen is made and

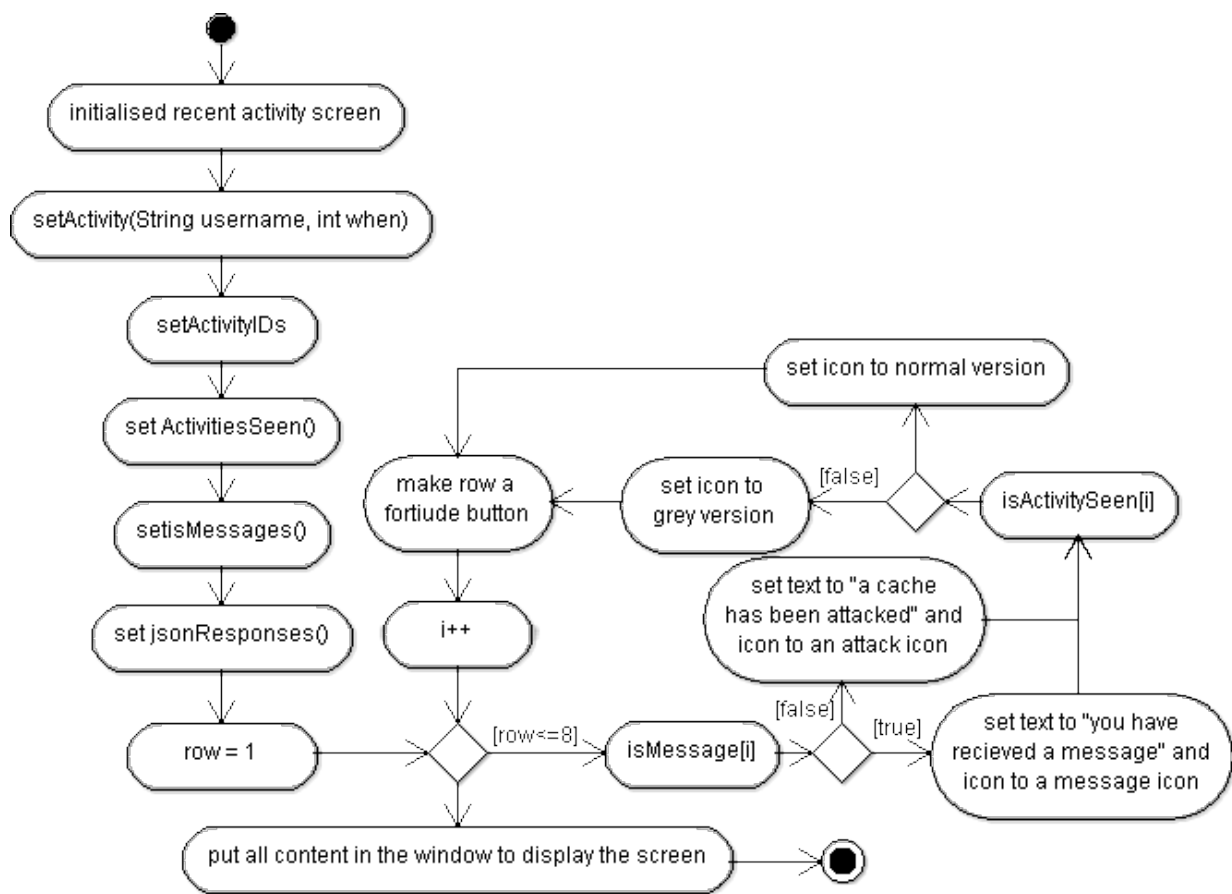


Figure 3.22: Activity diagram of the recent news list.

the button is pressed `User.getBlockedUsers().contains(String username)` must be called to decide which button is displayed and what action is taken. In the settings menu for messages a list of blocked users is displayed so whenever this screen is initialised, the `User` class is used to find the names of users who are blocked. When a user sends a message to a blocked user, the message will still be sent to and stored in the server, however the user will be shown a message box saying the message was unable to send.

There is a recent news screen which works much like the inbox screen. When it is initialised `setActivity(String username, int when)` from the `serverRequests` class is called to add the information needed from the server. It creates button rows that have different text and icons using this information. It also stores the JSON responses on the app so that when a row is clicked the user can go to the right screen. If the boolean corresponding to the row clicked in `isMessage` is false then the cache is extracted from the corresponding response using `get("cache")` and this and the response is used to display the post battle screen. If the corresponding value in the `isMessage` array is true then values for sender, subject and body are extracted and stored as the corresponding current message variables in the `Messages` class and used to view the message screen. This satisfies requirement 5.3.6: Activity recording.

Special caches will involve the `SpecialCacheUpdate` class that works much like the `IconUpdater` class. It is constantly running in the background, getting user's WiFi networks and checking if any have a MAC address of a special cache. If the user can connect to a particular MAC address, then if they are currently on the main screen, the screen will have added to it an icon that can be pressed, and when pressed the user will be shown the `SpecialCacheFoundScreen`. From here they can click to claim their prize, and if they can still connect to the network with the right MAC address and the reward is still available, they will be

displayed a message telling them they were successful, their account will be refreshed so the server knows not to use this special cache in the SpecialCacheUpdate for this user again, and the user's balance will be updated, which will satisfy requirement 5.2.2: Account transactions, and 5.2.21: Special event rewards. If they can no longer connect to the correct network they will be displayed a message saying they were unsuccessful. Either way they will then be taken back to the main screen.

After user actions that could change data, the data is refreshed which means it will be accurate so a user can see their account balance accurately at any time satisfying requirement 5.2.1: Account balance, and points can be added or removed from an account balance by other components of the system satisfying requirement 5.2.2: Account transactions. As all user actions involve displaying a message box requirement 5.2.14: Cache operation chronology will be satisfied because a message box disables the screen underneath, so the user can't do anything else until the server has finished processing the action.

This section will now take a look at the design of the server. When a request is made in general, the processes in figure 3.23 will occur. The APIManager class will receive a URL from an external source in a form such as /api/scout?uname=Metapyziks&session=0123456789abcdef. It will use the Request class to find the specific request class needed so in this case it would be the ScoutCacheRequest class. It will then use the method Respond in the specific request class to get a response that is changed into JSON form and sent back to the external source. However if a specific request class can't be found then the ErrorResponse class is used. The server is not going to involve threading so each task will be carried out one at a time to satisfy requirement 5.2.14.

Sequence Diagram

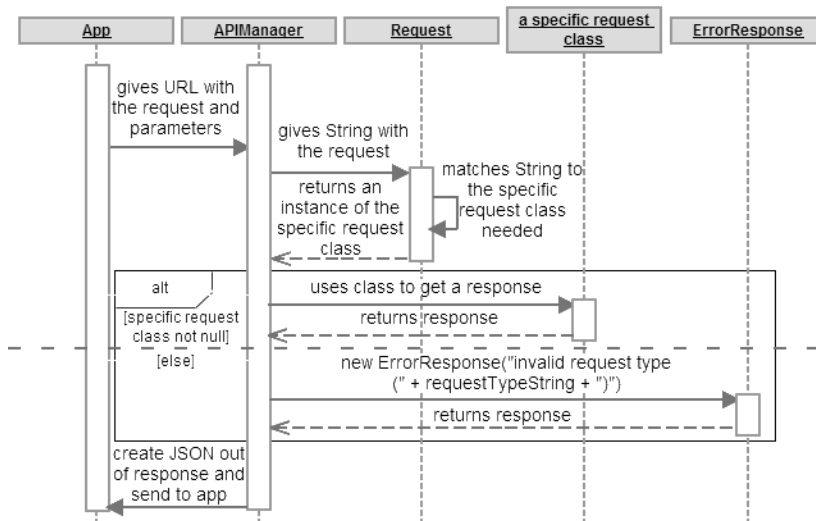


Figure 3.23: Sequence diagram of the general server API system.

requirement 5.1.3: Administrator Accounts) This account is placed in the database and used as a parameter in EmailValidationCode.create() along with the Enum EmailValidationType set to Activate. The code created is used to send an email to the user to activate their account, which will satisfy requirement 5.1.2:Account Activation. If it has got to this point in the method then null is returned so the Response is made using the Response method which gives a successful response rather than an error.

When resending an activation email it is the SendActivationRequest class that performs checks to satisfy requirement 5.4.1: User Communication. The account with the email address provided is selected from the database and used to resend an activation email in the same way as account registration. If an ErrorResponse hasn't already been returned then a Response will be like with account registration.

Sequence Diagram

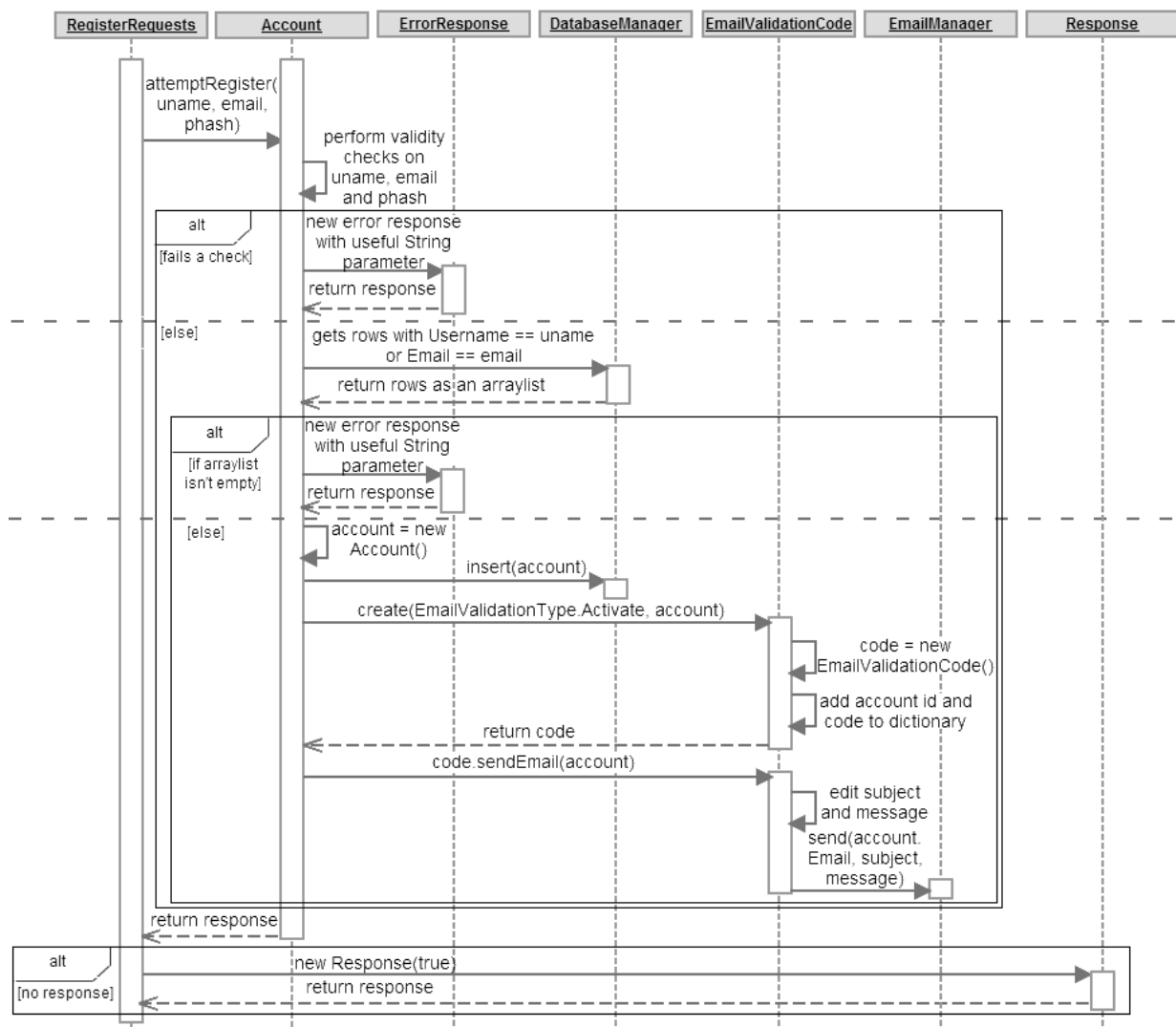


Figure 3.24: Sequence diagram of the user registration process.

When sending a password reset email, PasswordResetRequest let's EmailValidationCode perform the checks to satisfy requirement 5.1.8:Password Reset in the method AttemptCreate(). If it passes these checks an EmailValidationCode will be created and used to send an email to the user and a Response will be returned, else if the checks aren't passed an ErrorResponse will be returned.

Figure 3.25 shows how UserInfoRequest works, as you can see the checks are performed in the specific request class. If passed then the accounts and caches of the users wanted are found. The response is found using this list of Account and a list of Int corresponding to the number of caches the user has. UserInfoResponse uses these parameters to make a response in the form:

```
{"users": [{ "accounted": 0, "uname": "metapyziks", "regdate": 1347374370,
"rank": Unverified, "caches": 0 }, { "accounted": 1, "uname": "TestName",
"regdate": 1347381045, "rank": Verified, "caches": 23 } ], "success": true}
```

Sequence Diagram

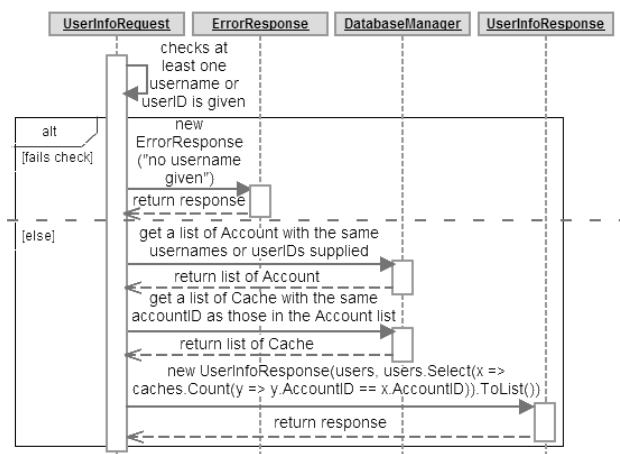


Figure 3.25: Sequence diagram of the user information request.

Sequence Diagram

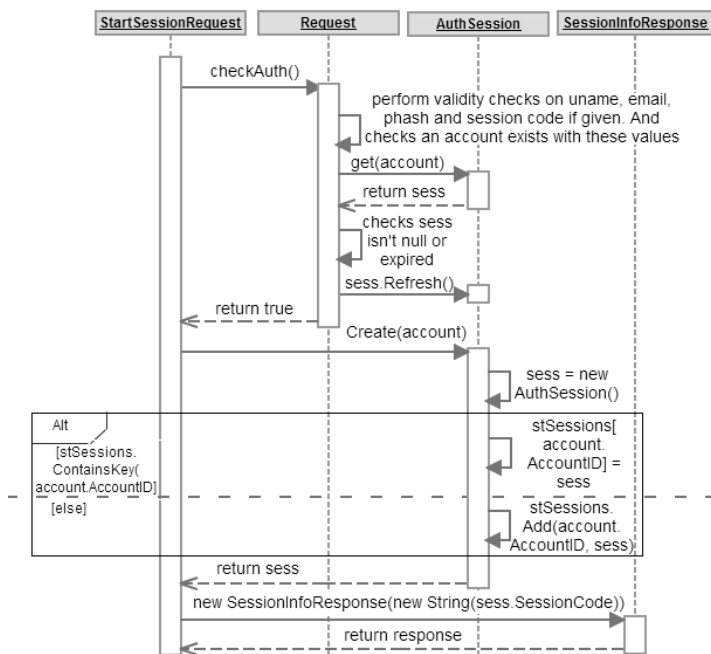


Figure 3.26: Sequence diagram of the session creation process.

InfoResponse with a code and success value. This satisfies requirement 5.1.5: User authentication.

The UserStatsRequest class works by checking in the same way as StartSessionRequest using CheckAuth(). If the checks are passed an instance of Player is created using Player.GetPlayer(account), which

The server creates a new session using StartSessionRequest. This performs all the necessary checks on the username, email, password hash supplied with the Request class, which will also get the account corresponding to the user supplied and check the account along with the SessionInfoResponse corresponding to the account. It checks expiration using the method IsExpired() in AuthSession. If any of the checks are failed a related ErrorResponse is initiated and fail is returned to StartSessionRequest which will not go on to do AuthSession.Create(account). If the checks are passed the instance of AuthSession is refreshed which will set the refresh time to the current time so it can be used in the IsExpired() method. True is returned to StartSessionRequest, a new instance of AuthSession is made with an AccountID, SessionCode, LastRefresh set to now and a Rank set to the rank of the account. The AuthSession is added to a dictionary of AuthSession as an update or new entry with a new AccountID. The AuthSession is returned to StartSessionRequest which is used to create a new Session-

selects the player in the database with the correct AccountID using Select() in DatabaseManager. If this player doesn't exist yet then a new Player is created with a balance of 0 and AccountID then inserted into the database using Insert() in DatabaseManager (see scouting figure). The caches the user owns are found using Select() with the caches AccountID as the account's AccountID. caches.Sum() is used to find the number of points stored in all the caches owned by the account holder and this along with the player Balance and cache Count is used to create a new UserStatsResponse. This satisfies requirement 5.2.1: Account balance.

To list a users' caches, UserCachesRequest is used, which uses CheckAuth() to find errors. If there aren't any errors then a list of Cache is found with the same AccountID as the Account of the specified user using DatabaseManager. The response including the caches' cacheid, ownerid, name, latitude, longitude and garrison is found by new CacheInfoResponse(caches).

To find a list of nearby caches, NearbyCachesRequest is used. This firstly find errors with CheckAuth() then if none are found it will check the latitude, longitude and radius can be parsed as a double and if not make a new related ErrorResponse. Else that latitude, longitude and radius are used for Cache.FindNearby() which uses DatabaseManager.Select() to find a list of Cache in the radius. The balance of the caches not owned by the user are set to -1 so the user can't know about enemy garrisons, satisfying requirement 5.2.4: Cache balance. The balance of non-player caches is set to -2 to make them distinct from other caches, and they will only be put in the response if the user hasn't defeated the cache in a certain amount of time, satisfying requirement 5.2.19: Scouting Non-Player Caches. Again CacheInfoResponse is used if there wasn't an ErrorResponse needed. This satisfies requirement 5.3.2: Nearby caches.

When a cache is scouted, the ScoutCacheRequest class is used. Checks's are made using checkAuth(), by trying to parse the Strings representing numbers (e.g. units and latitude) into Doubles, and by seeing if the numbers are in the right bounds e.g. positive or smaller than the user's balance. Tools.getDistance() uses trigonometry to find the distance between the user and closest cache. Tools.CoinToss used to calculate the survivors, it creates a random number and if this number is less than 0.8, true is outputted. This is done for every soldier in the scouting army and the number of trues outputted equals the number of survivors. The players balance is set to balance – units + survivors and this is updated in the database. A new ScoutCacheResponse is made with the CacheID, Scouts set to units, Survivors and Garrison set to cache.Balance. The response will include cacheid, ownerid, cache name, latitude, longitude, garrison and success. The response will be put in Activity, the list of responses in the corresponding Account class and updated in the database. This satisfies requirement 5.2.9: Cache scouting.

The PlaceCacheRequest class performs checks much like the ScoutRequest class, it does CheckAuth(), CheckLocation(), checks units are a Double, larger than the CachePlacementCost and smaller than the player's balance. CheckLocation() see's if the location hash supplied is the same as the location given which will satisfy requirement 5.2.24 anti-cheating measures. It also uses Cache.FindNearby() which uses as supplied minimum distance, latitude and longitude to find the caches within that distance, and PlaceCacheRequest uses this information to see if the user is far enough away from other cache to place a cache. If all of the checks are passed then a new Cache is made with an AccountID, Balance which is 4 less than units, Latitude, Longitude, Name which is randomly selected, CacheID which isn't supplied by PlaceCacheRequest and Nonplayer set to false. The Cache has an associated AccountID to satisfy requirement 5.2.3: Cache ownership, and a Balance to satisfy requirement 5.2.4: Cache balance, which is less than units to satisfy requirement 5.2.8: Cache placement cost. The players balance is updated and saved in the database, and the cache and activity is inserted into the database. A new CacheInfoResponse is made which includes cacheid, ownerid, name, latitude, longitude, garrison and success. This satisfies requirement 5.3.5: Cache placement.

As you can see the checks for attacking caches are done in the AttackCacheRequest class, and the response is created in the Cache class. The sum used for weight is based off an estimate of figures in pre-Saxon

Sequence Diagram

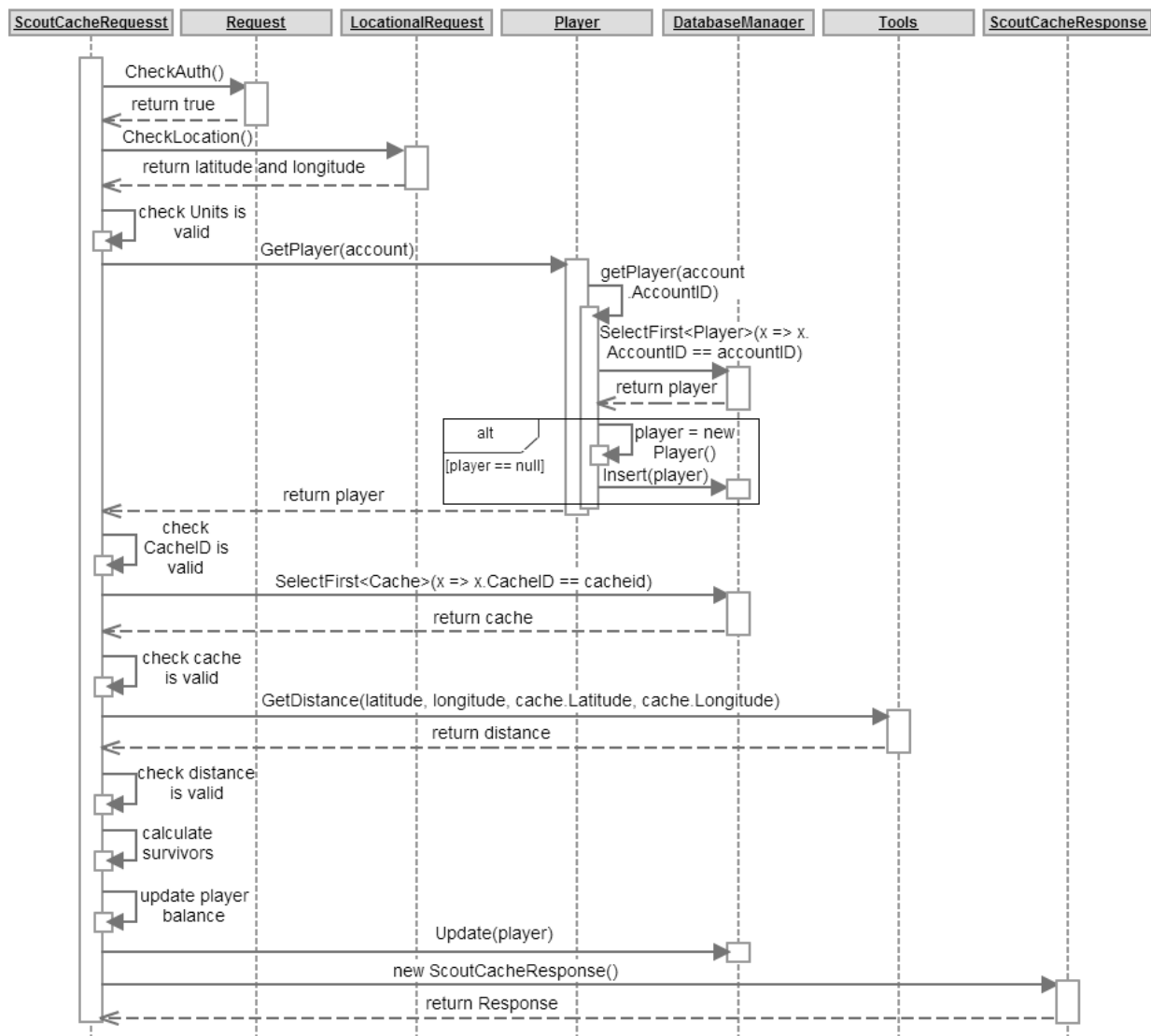


Figure 3.27: Sequence diagram of the cache scouting process.

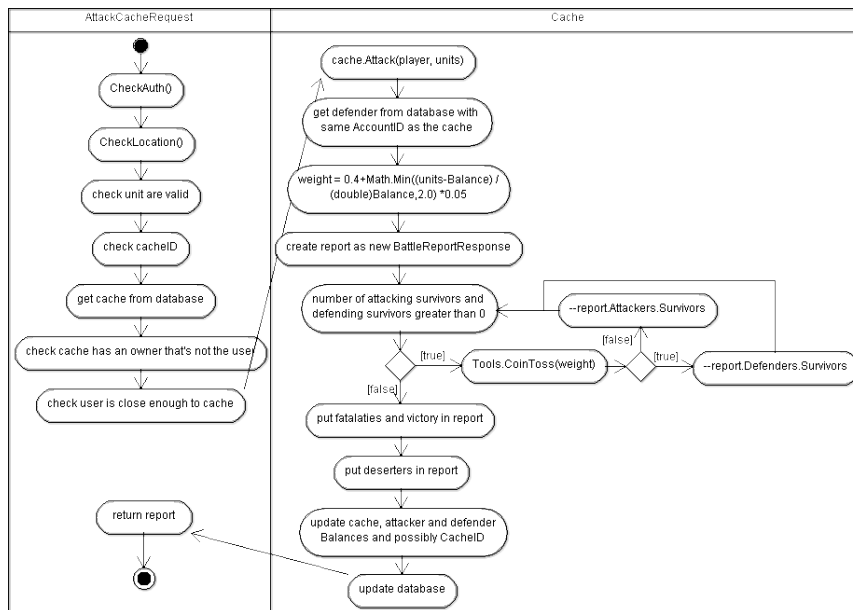


Figure 3.28: Sequence diagram of the cache attacking process.

Britain. The BattleReportResponse is initially set with the cache, AttackerID, DefenderID, Attackers made from a new BattleReportResponse.UnitReport with Initial and Survivors set to units, and Defenders made from a new BattleReportResponse.UnitReport with Initial and Survivors set to cache Balance. The fatalities are put in the report with `report.Attackers.Fatalities = report.Attackers.Initial - report.Attackers.Survivors` and a similar method for Defenders. `Report.IsVictory` is true if there are attacking survivors, in this case `report.Attackers.Deserters = 0` while the defending deserters are a small random proportion of the defending fatalities, the cache Balance becomes the surviving attackers plus the defending deserters and the caches AccountID becomes the attackers AccountID. However if `IsVictory` is false, there are 0 defending deserters, a small proportion of the attacking fatalities are deserters, the cache balance is the surviving defenders and the attacking deserters are added to the defenders balance. The response is added to Activity in the affected instances of Account and updated in the database. This satisfies requirement 5.2.10: Cache attacking, 5.2.11: Successful attack, 5.2.12: Successful defence and 5.2.13: Battle breakdown. Attacking non-player caches is very similar, the same request is used, the only difference is that the user's Account NonPlayerDictionary will be updated with the cacheID and time, the 'point reward' will be shown as the deserters, which there will be more of to make up for not getting the cache, and the cache balance won't change in the database. This satisfies requirement 5.2.18: Attacking Non-Player Caches.

When a message is sent the MessageRequest class is used, it checks the message body and subject aren't empty. It will create a MessageResponse and put it in Activity of the sender's and receiver's Account and updated on the database. It will check the receiver's Account to see if the sender is blocked and return an ErrorResponse if the sender is blocked else a Response is returned. This satisfies requirement 5.4.1: User Communication. The receiver will always get the message as it is when the activities are requested that the server will make sure that messages aren't received from blocked users. This is so that when a user is unblocked the messages will be received and in the correct order in relation to other activities. There will obviously be a request to block users too which will check the username or id supplied has an associated account that isn't the account of the user before adding the id to the Block list in the user's account. The possible responses are the ErrorResponse or Response. The request to unblock a user is the opposite to blocking users.

Messages and caches will be able to be reported through the MessageReportRequest and CacheRe-

portRequest classes. They will use the reporterid or username, message or cache id, and a description. They will check the cache or message hasn't previously been reported, and if it's a CacheReportRequest then that it's not the reporter's cache and it's not a non-player cache. The message id is searched for in the reporter's activity to find the message details, or the cache details are found using the cacheid. This information is put into a related Response and added as an AdminResponse in the database so administrators can see it on the website. The cache or message will be marked as reported and saved in the database and a Response will be returned. This satisfies requirement 5.2.22: Cache Reporting and 5.4.2: Communication Reporting.

In many requests the DatabaseManager and Account classes are used in conjunction with the specific request class, with this requirement 5.2.2: Account transactions and 5.3.6: Activity Recording can be fulfilled. There will also be a Request class for getting activities that uses the users Account ID or username, how many activities they wish to see, which set of they want, and what type. It performs checks (including removing messages from blocked user's from the response) and uses modular arithmetic to select the correct set of activities of the right type and return them in ActivityResponse. Requirement 5.2.5: Cache transactions is fulfilled in the same way as 5.2.2 but with Cache class, the request used to withdraw and deposit soldiers will also set the cache owner to null and update it in the database if all the soldiers are withdrawn. This will satisfy requirement 5.2.6: Cache Withdrawal.

The design of the website will now be discussed. Although not all pages are mentioned, the pages not mentioned will feature forms, checks and updates much like the pages which are examined in detail. This will satisfy requirement 5.4.3: Website.

The index page is made as followed, firstly the header is used to find information needed. It includes function.html which provides a list of methods used for checking such as isAdmin() and pluralify(). If the Account of the user is verified then the Player and Cache classes are used to find the player and their caches. If the account has the rank admin or higher the admin panel is displayed along with the header everyone can see. If the session isn't null (ie isSignedIn()) then the side bar is personalized, and if the player and caches aren't null it is personalized further with the number of caches and the players balance. If the account is verified and there is a session then caches and news plus links to further information, caches and news are displayed in the sidebar. This sidebar will satisfy requirement 5.4.5: Viewing Owned Caches and 5.3.6: Activity Recording. The links to caches will display pages that satisfy requirement 5.4.6: Viewing Cache Details.

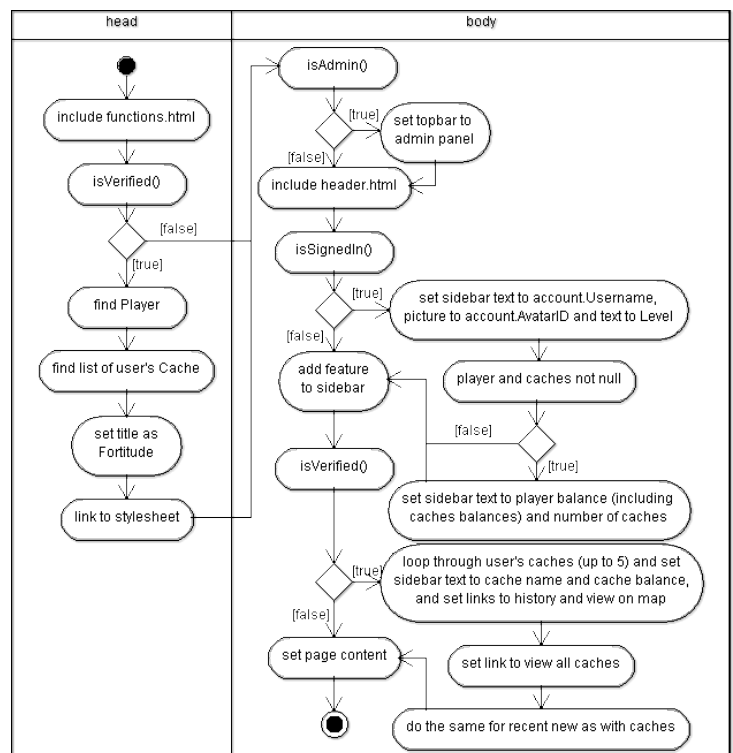


Figure 3.29: Activity diagram of the index page construction.

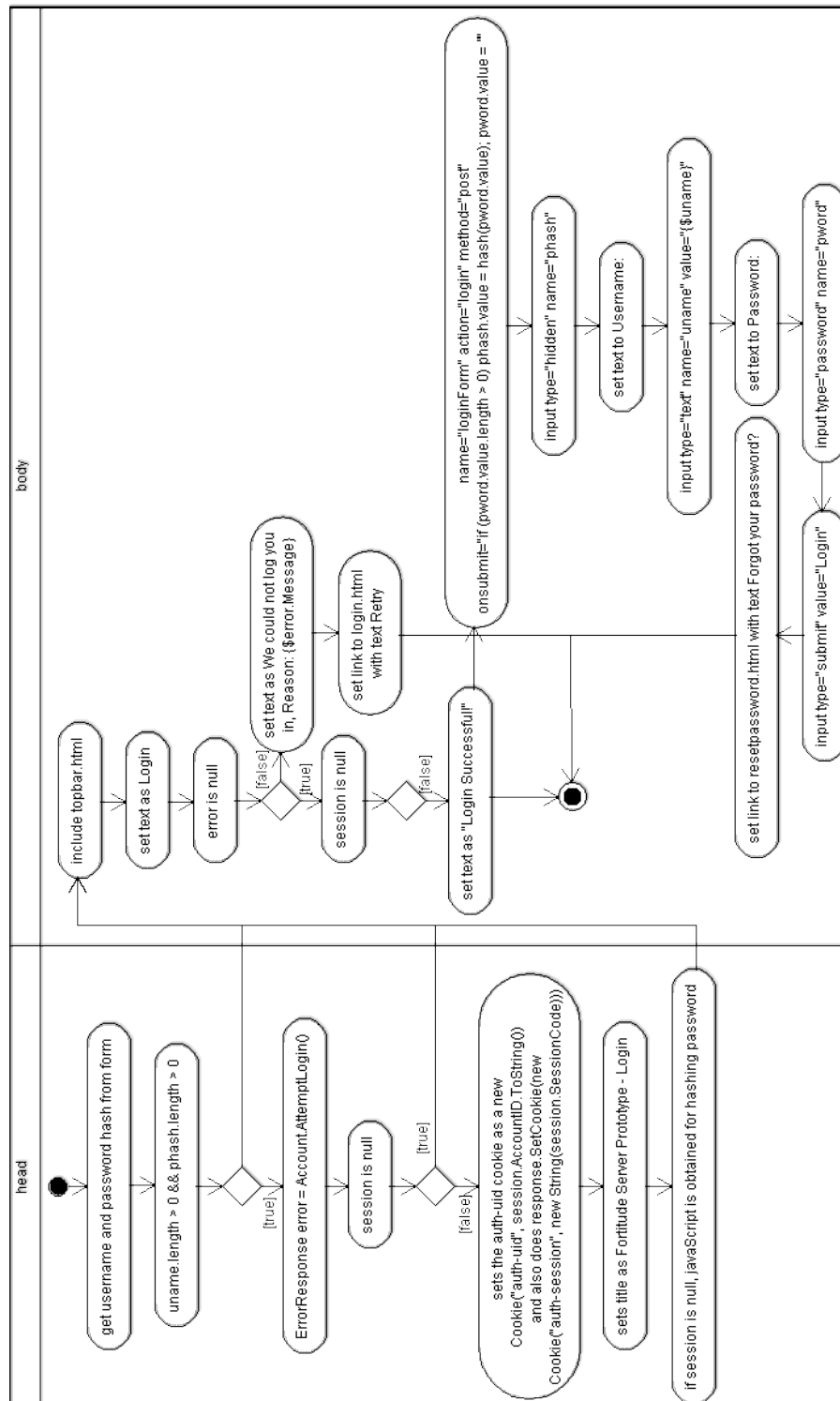


Figure 3.30: Activity diagram of the login page construction.

When a user logs in with the website login.html is used. It's head uses the username and password in the form to attempt to login, which will output a session. If the session outputted by Account is null a new session is made using cookies. The title is set and if necessary JavaScript is loaded for hashing passwords. The top bar is added, the subtitle set to Login. If there's an error then it is explained and a link is displayed to try again. If the session is null a form is produced for the user to enter their username and password, which will be hashed. A link is also provided in the case that the user forgets their password. If the session is not null then text will be displaying saying successfully logged in. This will satisfy requirement 5.4.4: Website Authentication.

When a user clicks a link in an email to activate their account activate.html is used. In the head, there will be AttemptActivate(get["email"], get["code"]) and the title will be set to Fortitude Server – Account Activation. In the body, a sub-title will be set as Account Activation. If AttemptActivate outputted an error that wasn't null then the user is explained the error, if not the user is told their account has been activated and they can now log on. A link will be provided to the Home page at the bottom of the body. This will satisfy requirement 5.1.2: Account Activation.

Admins can place a cache with placecache.html. In the head this checks whether this session is null and the rank of the session. If the session is null or the rank less than administrator the user is redirected to the error page. Else the account corresponding the username submitted is found and checked, if anything fails an error message is set. Else a new Cache is created and updated in the database. The title is set to Place Cache and a form is created to input username, latitude, longitude and units. If success is true then text is displayed saying where the cache was placed else if there was an error, the message is displayed. This satisfies requirement 5.2.16: Administrator Placed Caches. placenon-playercache.html is very similar except the form will only include inputs for position and units. The cache created and added to the Server will have the AccountID as the administrator's AccountID. placespecialEvent.html is also similar, in the head it will be checked if the supplied MAC-address is valid and the number of user claiming the reward and the reward is greater than 0.

These will be in the form in the body. Then the special event cache will be placed in the database. This satisfies requirement 5.2.20: Special Event Placement.

Administrators can delete caches using deletecache.html, this will redirect the user to the error page if their session is null or rank not high enough. It will use the cacheid in the form to get the cache from the database, which will be checked to exist then removed from the database. If it doesn't exist then the error String will reflect this. Text is set to Delete Cache and a form is created for the user to submit Cache ID. If success is true the user is told which cache was successfully removed by name and cacheid. However if there was an error text is set to this error. This satisfies requirement 5.2.23: Administrator Cache Deletion. deleteAccount.html works in a very similar way, it is an admin only page and when the account is removed from the database, all the caches with the accountID are found and removed from the database too. This satisfies requirement 5.1.7: Administrator Account Removal. Users can also delete accounts through the website then the email they will receive from an administrator. Once the link in the email is clicked on the users' caches as well as account will be removed from the database.

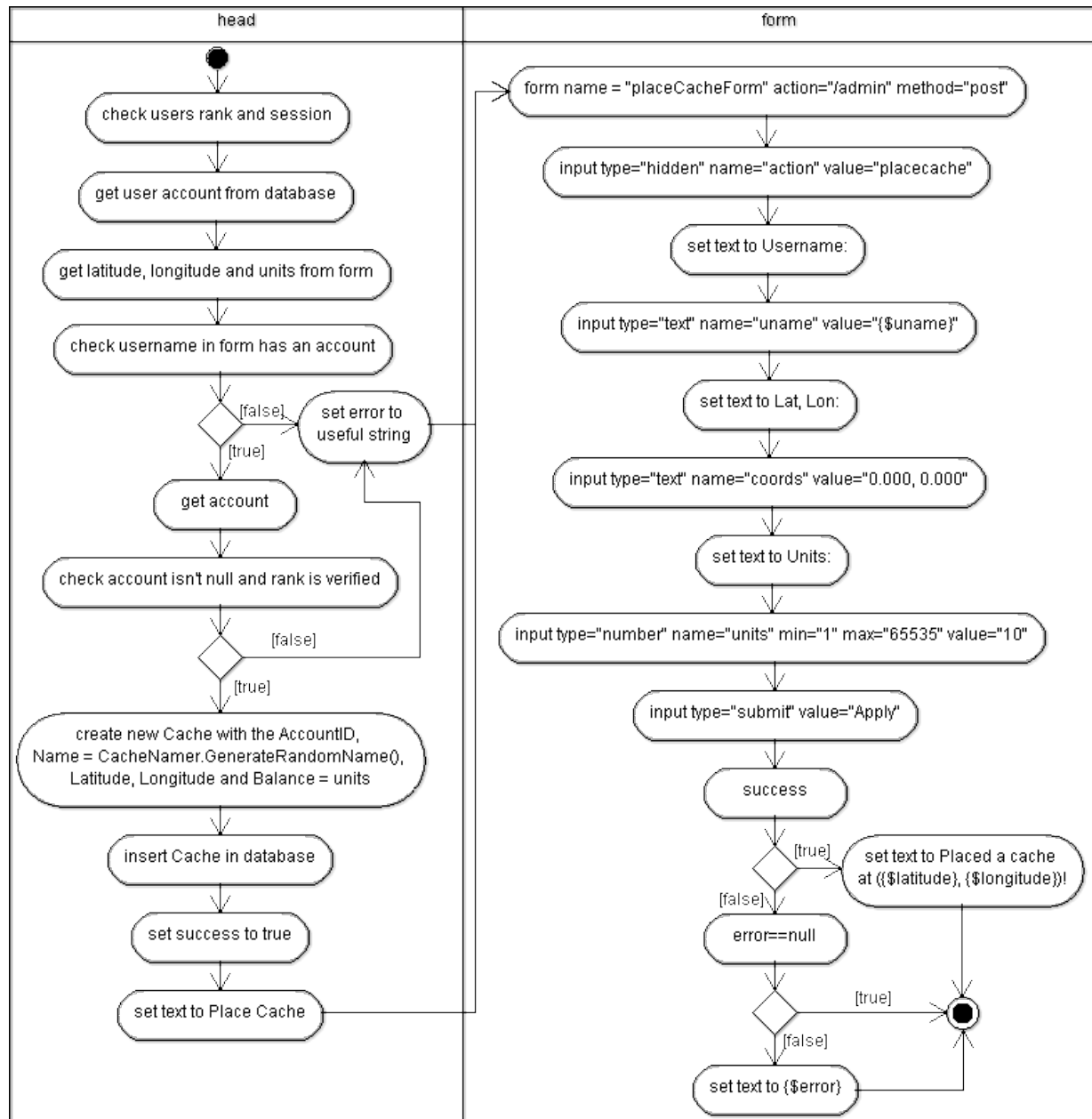


Figure 3.31: Activity diagram of the administrator cache placement process.