3 Element Descriptions

3.0 Architecture Overview

3.0.1 Top Level Design

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 many client applications that handle user interaction and present information through a graphical user interface. A requirement of the application was also to provide a website as a second source of 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.

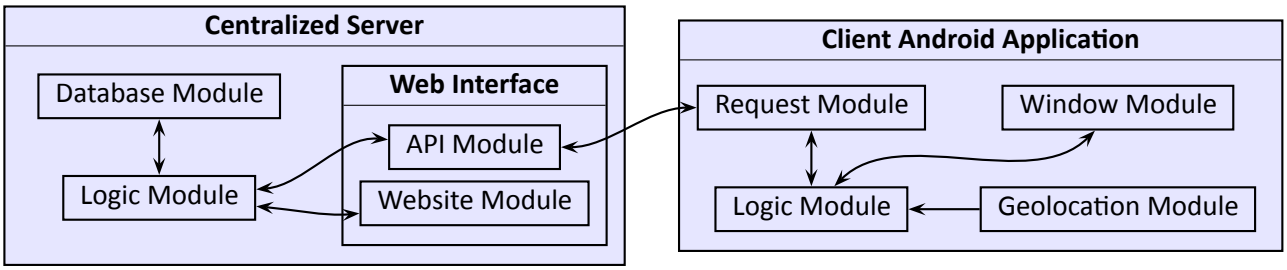

Figure 1: Data flow of the top level of the system, showing the application and server as clearly separated entities with internal modules that encapsulate distinct functionalities of the system.

As *Figure 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 the other modules in 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 Server.

On Unix systems, the server will be using the Mono CLR implementation, and will therefore have to use an alternative DBMS such as 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.

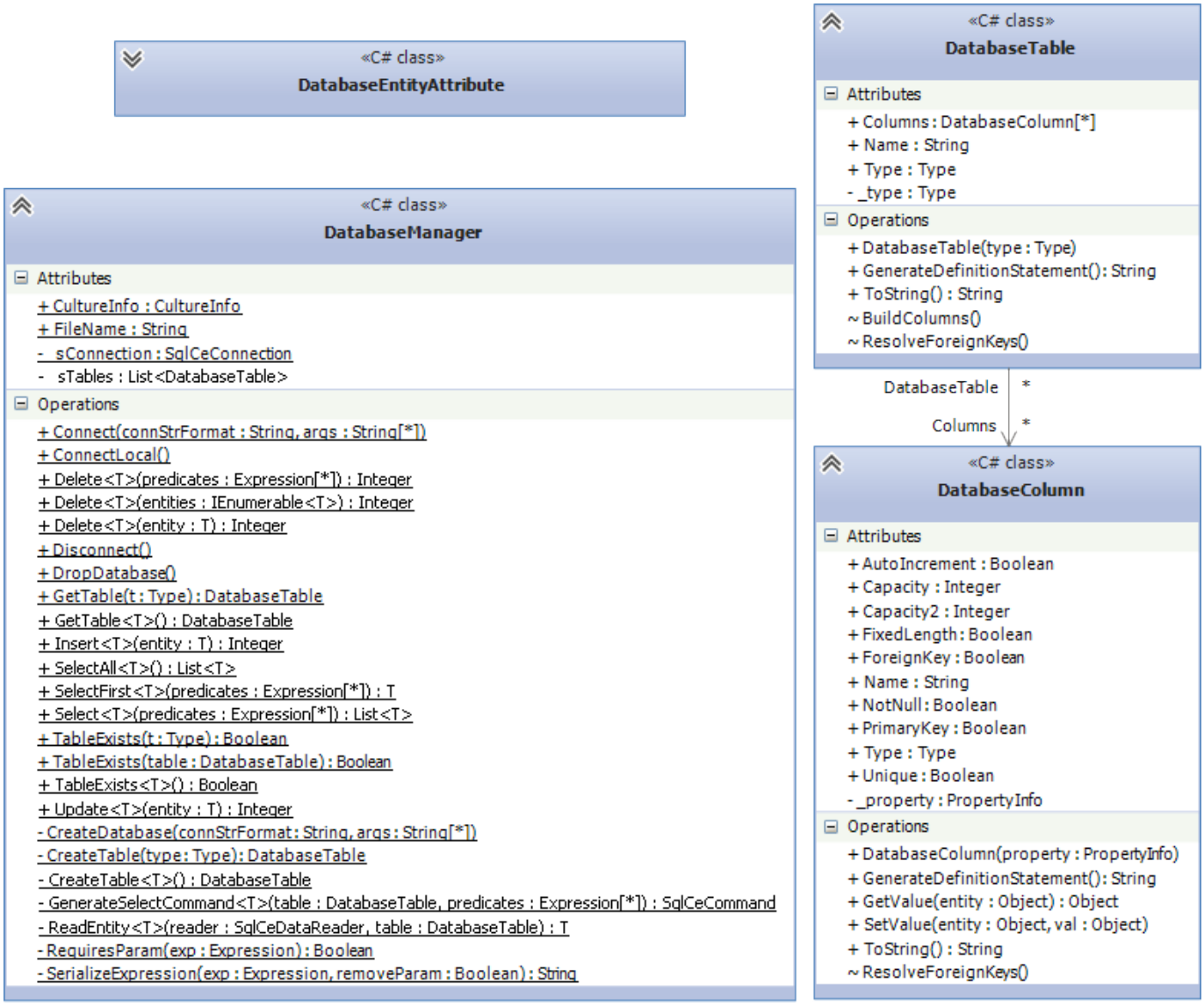

Figure 2: Class diagram for the main Database Module classes.

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.

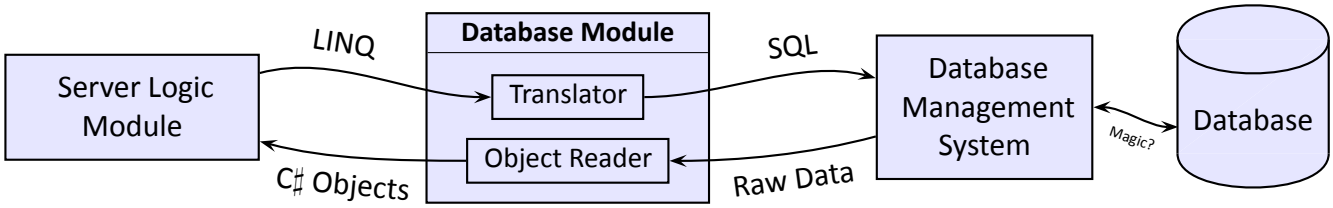

Figure 3: Diagram showing the data flow of the Database Module during a *SELECT* statement.

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.

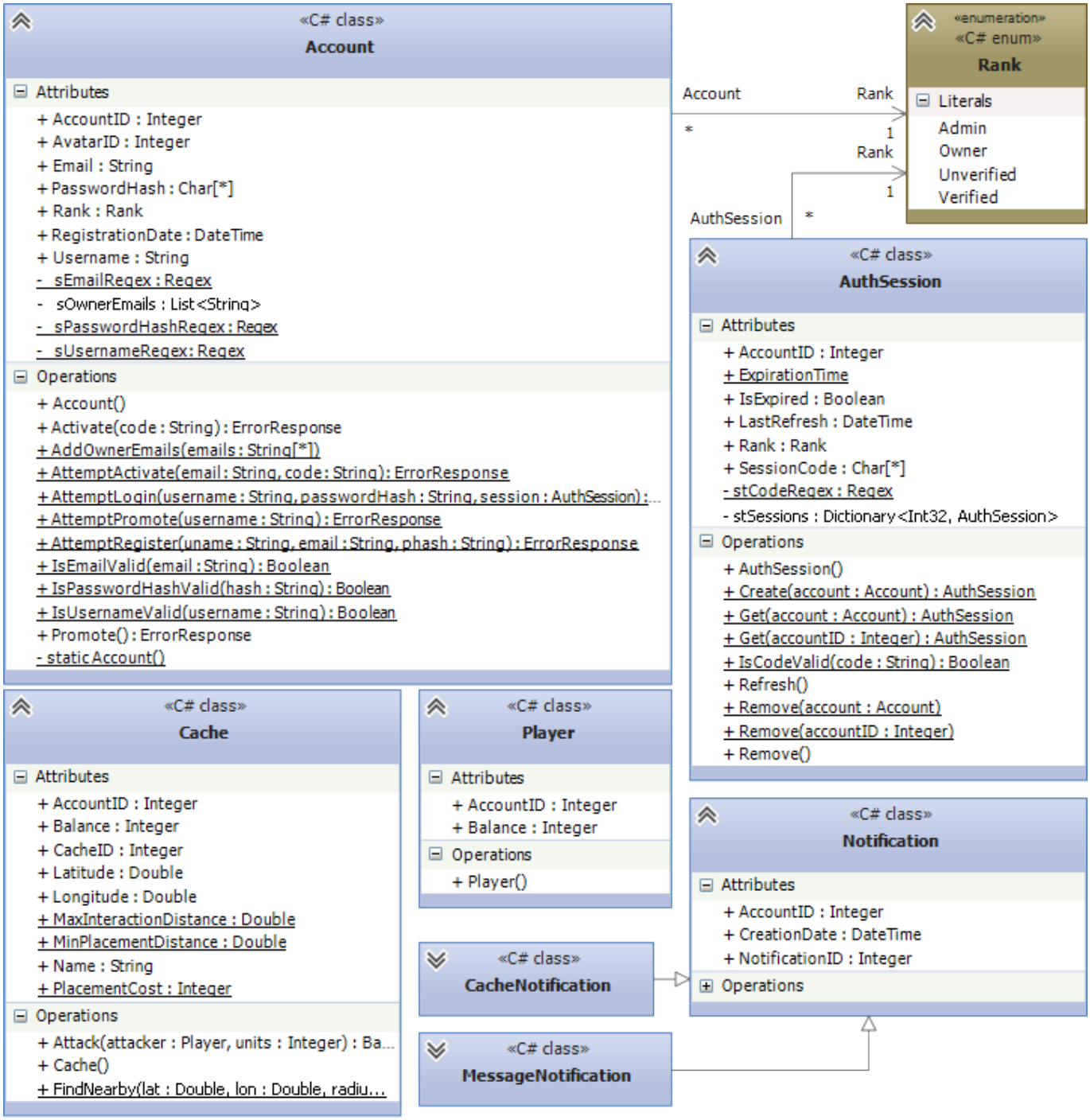

Figure 4: Class diagram for the main server Logic Module classes.

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.

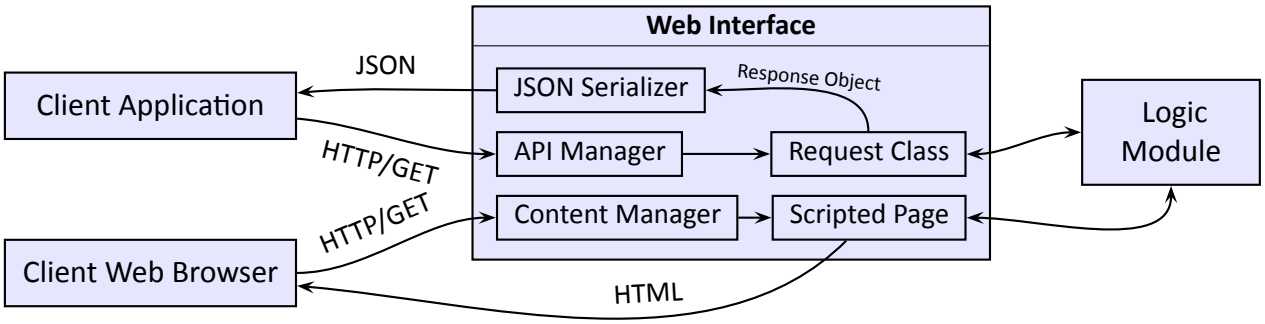

Figure 5: Diagram showing the data flow of the *Web Interface* module group.

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 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.

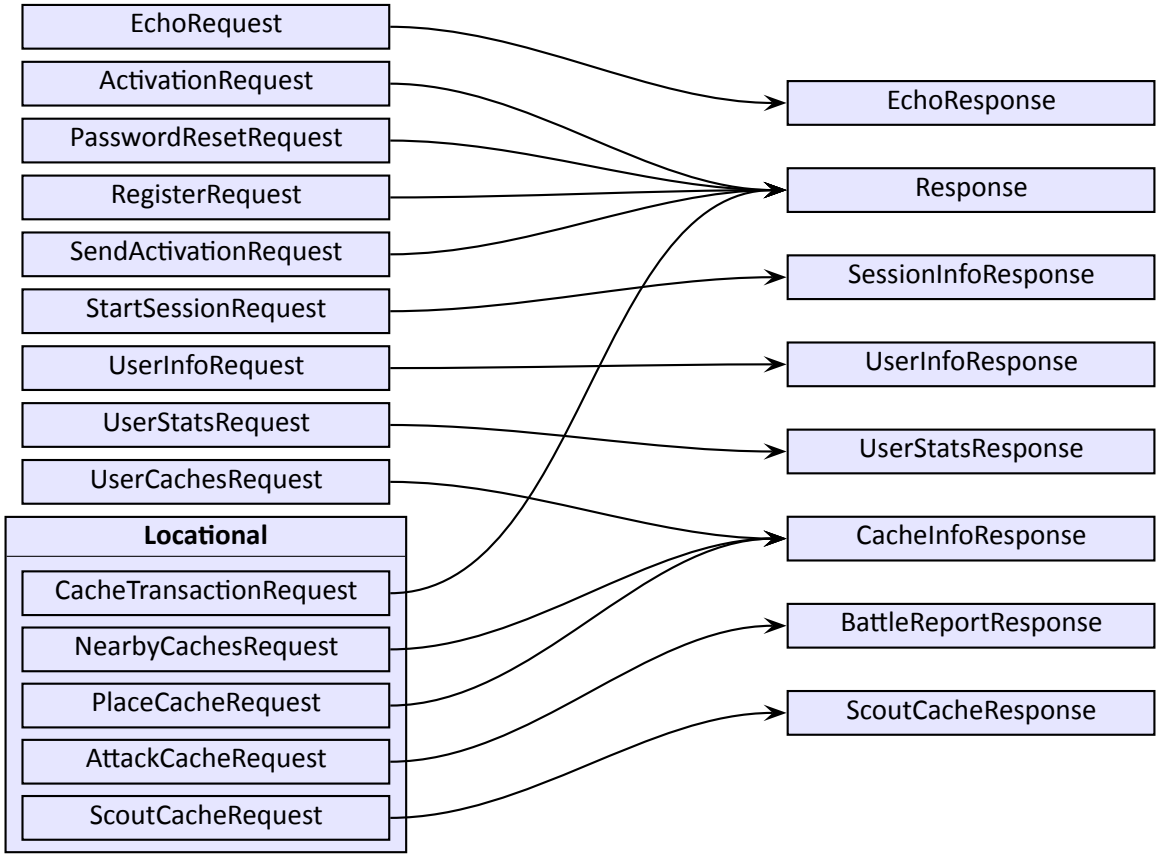

Figure 6: A list of each *Request* and *Response* type, and the relationships between them.

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 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.

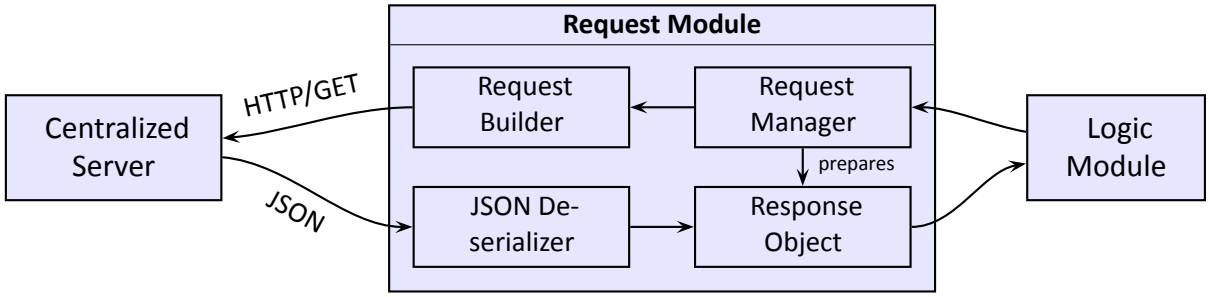

Figure 7: Diagram showing the data flow of the client application *Request Module*.

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.

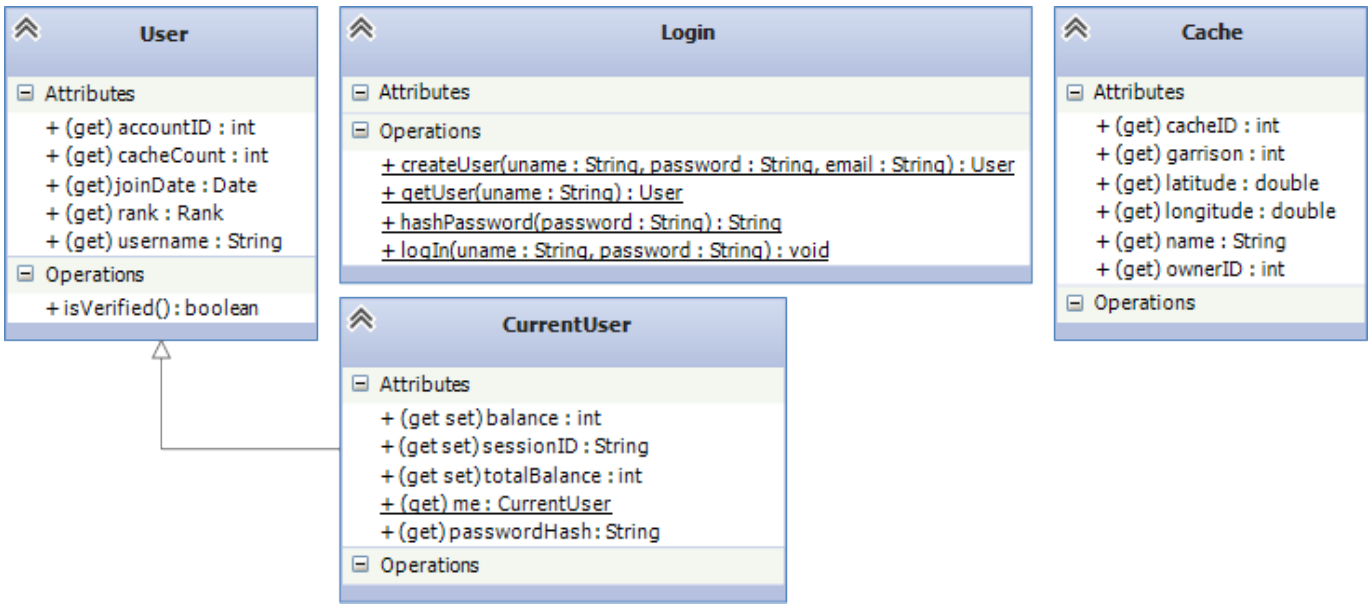

Figure 8: Class diagram for the main client Logic Module classes.

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 were 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 ``windows'', which are individual views that may be displayed to the client. Examples of windows are the login view, main map display, and the cache attack menu. All windows 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.