Carleton University

cuCare

System Design Document

Team: The Four Puppeteers
Sergey Vershinin (Team Lead)
David Carson
Devin Denis
Mike Yuill

Submitted to:
Dr. Christine Laurendeau
COMP 3004 Object-Oriented Software Engineering
School of Computer Science
Carleton University

FINAL
2012-11-22

Table of Contents

1 Introduction 1
1.1 Project Overview 1
1.2 Document Overview 2
2 Subsystem Decomposition 4
2.1 Phase #1 Prototype Decomposition 4
2.2 System Decomposition 7
2.3 Design Evolution 13
3 Design Strategies 17
3.1 Hardware / Software Mapping 17
3.2 Persistent Data Management 19
3.3 Design Patterns 22
4 Subsystem Services 24
5 Class Interfaces 41
Appendix I – Traceability Reference 46

Table of Figures

Figure 1 - Prototype #1 - Subsystems 4
Figure 2 - Prototype #1 - Classes 6
Figure 3 - cuCare - Subsystems 7
Figure 4 - cuCare - Classes 8
Figure 5 - cuCare - Components to Nodes Mapping 17
Figure 6 - cuCare - Subsystems to Components Mapping 18
Figure 7 - cuCare - Object Model 19
Figure 8 - cuCare - Persistent Data Management - ER Diagram 21
Figure 9 - cuCare - Persistent Data Management - Table Schema 21
Figure 10 - cuCare - Subsystem Services 24

1 Introduction

The Four Puppeteers Team proudly presents this cuCare system design document. This section will provide brief overviews of the cuCare project itself and the system design elements as they are presented in this document.

1.1 Project Overview

cuCare is an electronic records application designed to provide a medical practice with the means of leveraging modern information technology to streamline record management and encourage paperless records keeping.

The system design proposed herein is based on the requirements analysis submitted for your review on October 18, 2012.[1] The chief cuCare system features enabled by the proposed design include:

- Two primary system user types: physicians and their administrative staff. System administrator user accounts are also supported, explicitly for purposes of application configuration and user account management.
- Searchable patient records that contain each patients' contact, billing, and health information.
- Consultation management features, including setting the time and date of a new consultation, entry of pre-consultation data, in-consultation notes, as well as ability to create requests for follow-up care such as prescription renewal or specialist consultation, among others.
- An automated follow-up request tracking system to enable the practice to notify its patients if a request for follow-up care has not been confirmed as completed.
- A reporting facility able to provide a number of different report types including listings, aggregate metrics, and rankings of various patient record elements.

The system is based on client-server architecture, enabling simultaneous use of up to four clients connected to a single server. The server will feature a SQL-based relational database component enabling shared persistent data storage, accessible by the cuCare client applications via a TCP/IP network connection.

The cuCare client application will provide two separate interfaces, determined by the user type at system log-in. The medical staff interface will feature patient health and consultation scheduling system. The system administration interface will enable user management and configuration of the automated audit server process.

Designed to run on a Linux operating system, cuCare features a client application with an easy to use and intuitive graphical user interface (GUI). The cuCare system applications are being developed in C++, using the Qt Creator SDK and SQLite database library.

[1] This document covers the design of the entire cuCare system, including functionality that will not be present in the second cuCare prototype. Namely, system administration, user account management, and the reporting facility will not be implemented.

1.2 Document Overview

This system design document consists of four major sections including subsystem decomposition, design strategies, subsystem services, and class interfaces. This section provides brief overviews of each of these sections and the subsections contained therein.

Subsystem Decomposition

The subsystem decomposition section includes descriptions of the subsystems that made-up Prototype #1, a complete decomposition of the entire cuCare system, as well as a discussion of the differences between the two.

The first two sub-sections include a UML component diagrams detailing the dependencies between the subsystems of cuCare, as well as UML class diagrams depicting the classes included in each subsystem. Also provided are the brief descriptions of the purpose and function of each of the subsystems identified in the component diagrams.

- Phase #1 Prototype Decomposition
  This sub-section provides a subsystem decomposition of Prototype #1 that was developed using an agile development methodology, as it was submitted for your review on November 6, 2012.

- System Decomposition
  This sub-section features a subsystem decomposition of the entire cuCare system, including portions of the system that will not be implemented in Prototype #2.

- Design Evolution
  This sub-section explains the differences in the design of the two prototypes, and provides a discussion of the rationale behind the proposed changes.

Design Strategies

The design strategies section presents the hardware to software mapping of system components, a summary of the strategy employed to provide persistent data management, as well as an overview of how design patterns were employed in the proposed cuCare system design.

- Hardware/software mapping
  This sub-section describes how the cuCare subsystems are mapped onto software components, which in turn are mapped onto hardware nodes. The section provides a UML deployment diagram and a table of subsystem to component mappings to illustrate the approach.

- Persistent Data Management
  The persistent data management sub-section provides a description of the strategy for storage of persistent data within the cuCare system, and includes the rationale for the proposed approach of using an SQL-based relational database.

- Design Patterns
  The design patterns sub-section cites the instances where the proposed system design makes use of established software design patterns.

Subsystem Services

This section provides a description of services offered by each subsystem of cuCare. For each service a description of operations is provided, specifying the class to which each operation belongs. The entire set of system services is depicted by a component diagram using the UML ball-and-socket notation.

Class Interfaces

The class interfaces section provides a description of the classes involved in providing operations for each service. For each class a UML class diagram is provided, specifying its attributes and the operations involved in the service.

Note on Traceability:

Throughout the document, the reader will find references to the relevant cuCare use case numbers, as identified in the requirements analysis phase of the project. For ease of reference, a summary table of use cases has been appended to this document (see Appendix I).

2 Subsystem Decomposition

This section presents descriptions of the subsystems that made-up Prototype #1, a complete decomposition of the entire cuCare system, as well as a discussion of the differences between the two.

2.1 Phase #1 Prototype Decomposition

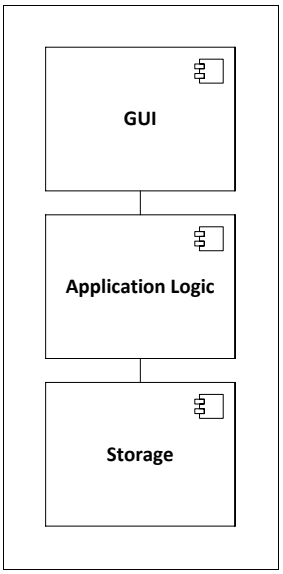


Figure 1 - Prototype #1 - Subsystems

The first prototype of the cuCare was developed using an agile development methodology, and submitted for your review on November 6, 2012. This section describes the design of Prototype #1 including its subsystems and constituent classes.

As per the requirements analysis document, Prototype #1 was supposed to be implemented as a client-server system. As such, the design called for a Communications subsystem in addition to the subsystems presented in Figure 1. However, having run into difficulty with the implementation of client-server communications, the submitted Prototype #1 ended-up using the design depicted here.

As per Figure 1, the first prototype of the cuCare includes three subsystems: the Graphical User Interface (GUI), Application Logic, and Storage. Each of the subsystems is described below. Please refer to Figure 2 for the Prototype#1 class diagram of the system, situating the classes in their respective subsystems and illustrating inter-class dependencies.

GUI

The GUI subsystem provides a graphical user interface for the cuCare client application. The GUI uses services of the Application Logic subsystem for data retrieval, storage, and access control. The GUI subsystem functionality is implemented using the following four classes:

- LoginWindowDialog: application access control; requires input of a valid username
- MainWindow: access to patient data, including physician consultations, follow-up requests, and contact information
- PatientSelectDialog: a searchable listing of patients registered with the system
- FollowupSelectDialog: a dialog prompt for selection of a follow-up request type

The majority of GUI classes, with the exception of FollowupSelectDialog, use services provided by the MasterController class. MainWindow and PatientSelectDialog also make use of the Model library.

Traceability	All use cases between (and including) UC-100 and UC-460, UC-800, UC-820, UC-830, UC-840

Application Logic

The Application Logic subsystem provides the GUI subsystem with data services and uses the storage and retrieval services provided by the Storage subsystem.

The MasterController class provides in-memory storage and management for all entity data. Data retrieval and storage operations (including user verification) are carried out via requests made to services provided by the Repository class.

The Model library provides entity storage classes, as well as Filter classes used to facilitate inter-process communication (IPC). For more information on the Model library please refer to Figure 2 below.

Traceability	All use cases between (and including) UC-100 and UC-460

Storage

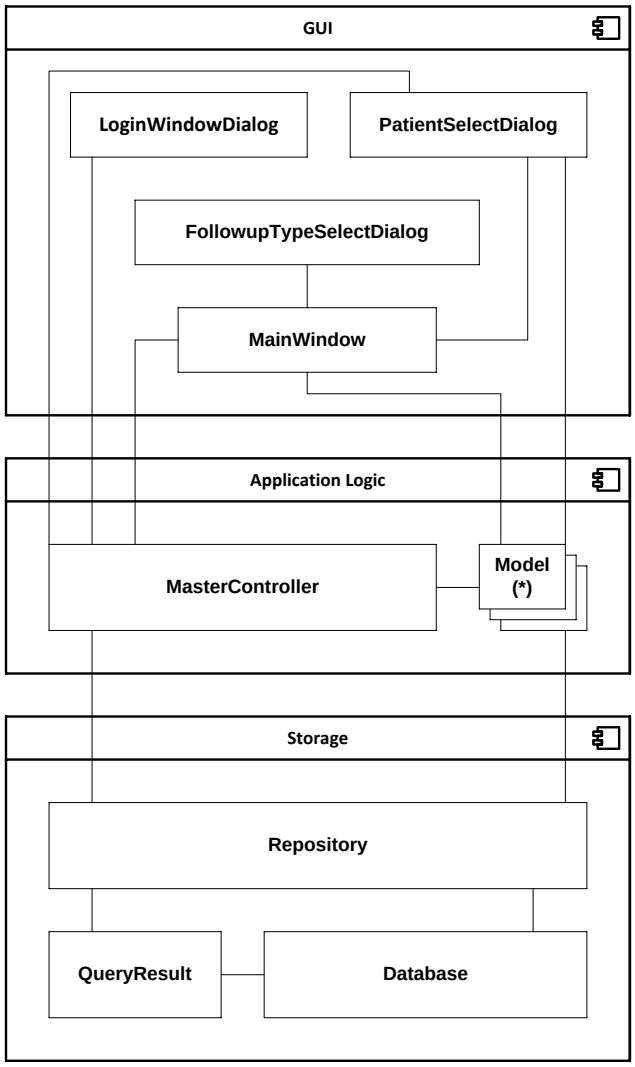
The Storage subsystem provides persistent data storage and retrieval services via an implementation of a SQL-based relational database[2], while abstracting away the use of SQL statements. The subsystem includes the following classes:

- Repository: provides the interface to the Storage subsystem that is used by the Server Controller subsystem; interacts with the Database class (by means of an object pointer instantiated in the constructor) to perform data storage and retrieval operations
- Database: implements an interface to the relational database (SQLite3, imported as a library), providing database access hiding the details specific to the selected database implementation
- QueryResult: wraps a vector of strings vectors, representing the rows and columns of the table returned by Database, and provides methods to simplify access to query result data

Traceability	All use cases between (and including) UC-100 and UC-460

[2] Relational functionality is implemented via SQLite v3. The SQLite database does not run as a seperate process. Instead, the database is a file that is accessed through library functions provided by the SQLite3 C++ API.

Figure 2 - Prototype #1 - Classes



(*) – The Model library contains the following entity storage classes:

- Address
- AdminAssistant
- Consultation
- ContactInfo
- Date
- Followup
- HealthCard
- MedicalTest
- MedicationRenewal
- Patient
- Physician
- Referral
- ResultantFollowup
- ReturnConsultation
- SysAdmin
- Time
- User

Also included are the following Filter classes:

- ConsultationFilter
- FollowupFilter
- MedicalTestFilter
- MedicationRenewalFilter
- PatientFilter
- PhysicianFilter
- ReferralFilter
- ResultantFollowupFilter
- ReturnConsultationFilter
- UserFilter

2.2 System Decomposition

This section presents a subsystem decomposition of the entire cuCare system, including portions of the system that will not be implemented in Prototype #2.

System Decomposition Diagrams

This section presents the UML component diagram illustrating the subsystems that make-up the cuCare system as well as their respective dependencies. Also included are class diagrams that depicting the constituent classes and their dependencies for each one of the proposed subsystems.

Figure 3 - cuCare - Subsystems

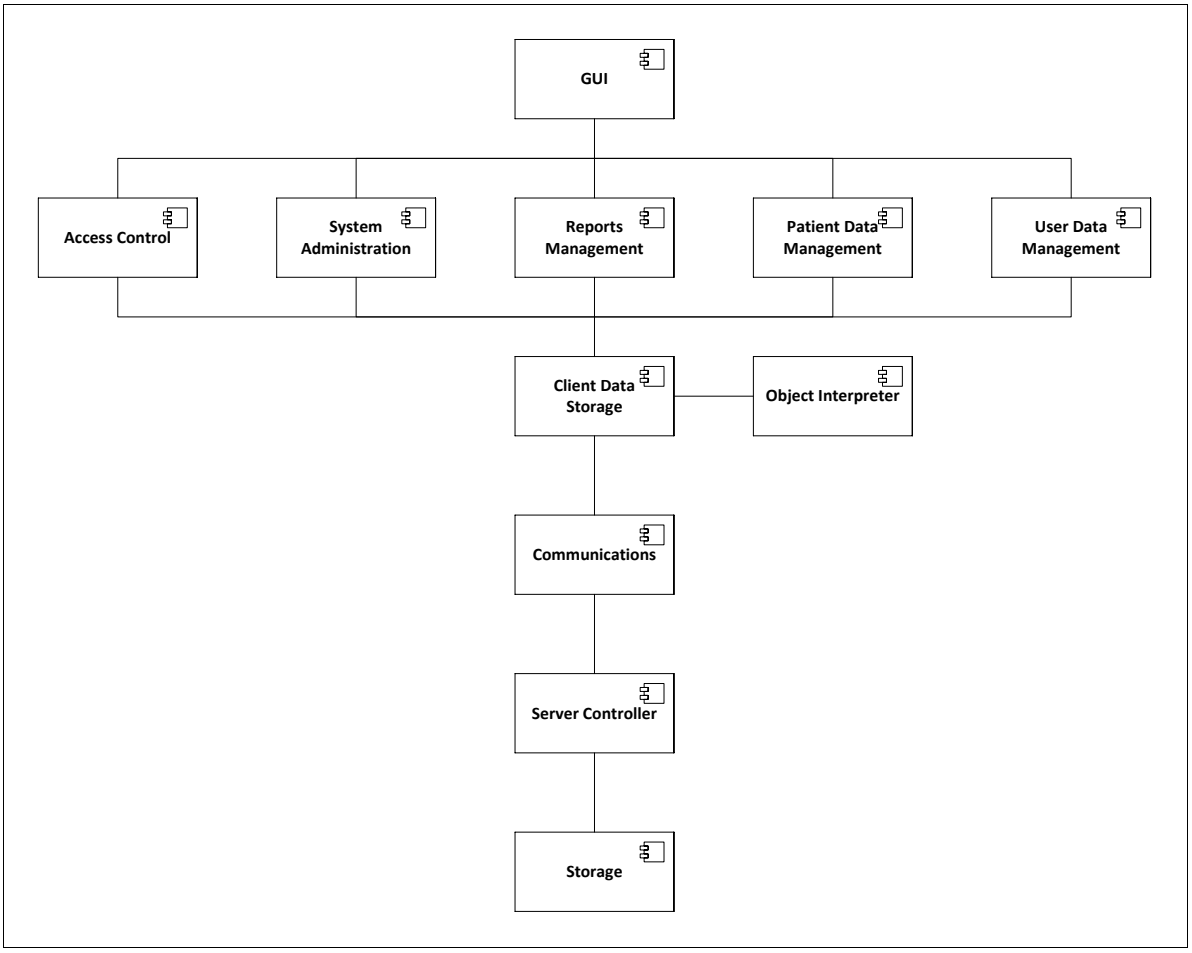
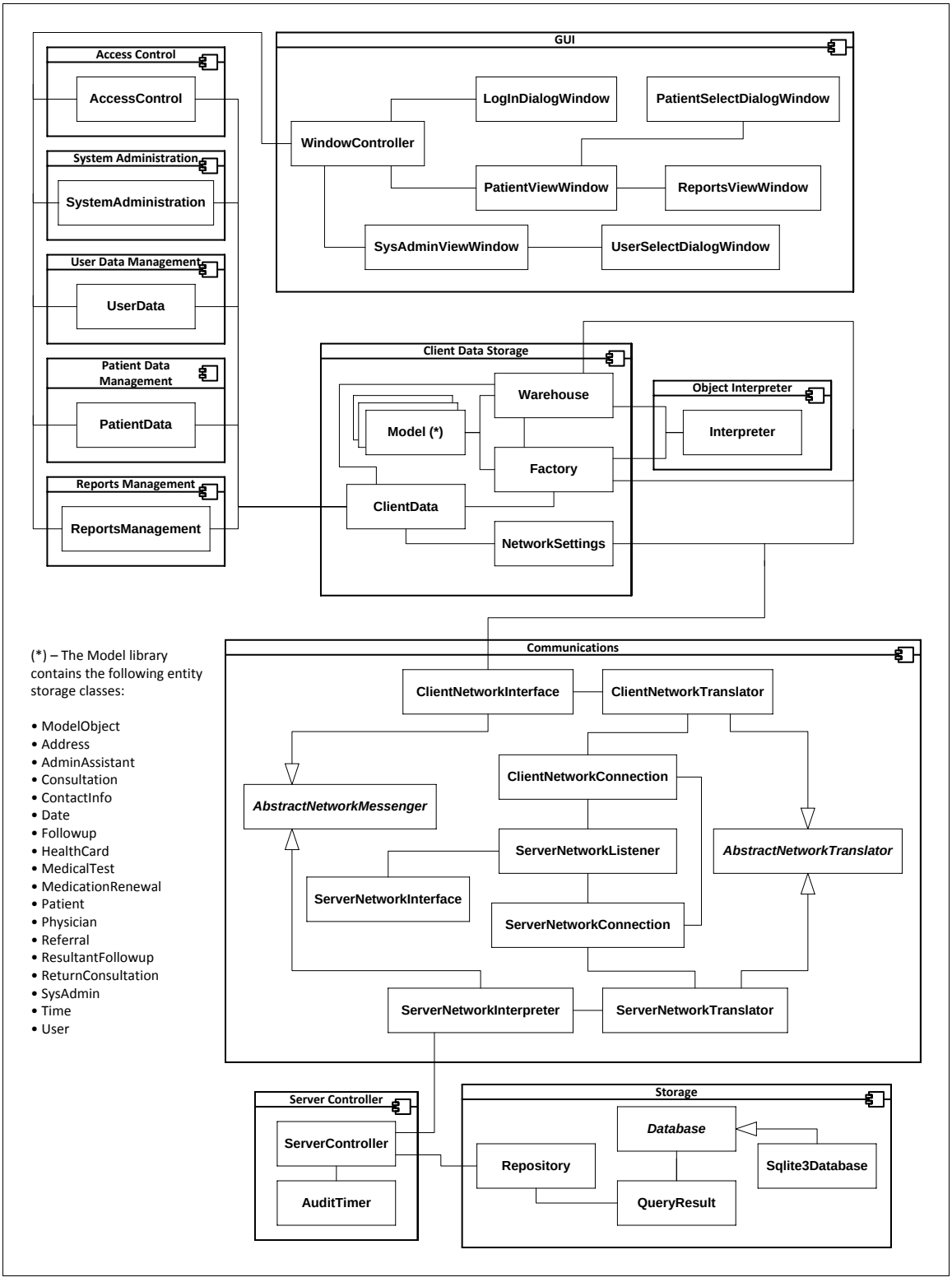


Figure 4 - cuCare - Classes



Subsystem Descriptions

Below, the reader will find descriptions for each of the proposed cuCare subsystems, their constituent classes, and other pertinent detail concerning the subsystems' implementation.

GUI

The GUI subsystem provides access to the system for the end user using a graphical interface. The interface is comprised of five different views: one for logging in to the system (LoginDialogWindow), one for viewing and/or modifying a patient and their associated data (PatientViewWindow), one for selecting a patient to display (PatientSelectDialogWindow), one for generating and viewing reports (ReportsViewWindow), one for viewing and/or modifying a user and their associated data, or for changing system parameters (SysAdminViewWindow), and one for selecting a user to display (UserSelectWindowDialog).

- WindowController: responsible for creating and destroying new instances of the LoginDialogWindow and the PatientViewWindow and displaying them appropriately as users log in and log out of the system
- PatientViewWindow: responsible for creating instances of PatientSelectDialogWindow when the user wishes to select a new patient, as well as instances of ReportsViewWindow when the user wishes to generate or view reports.
- SysAdminViewWindow: responsible for creating instances of UserSelectDialogWindow when the user wishes to select a new user.

Any operation requiring information to be stored on or retrieved from the server is handled by the WindowController class that uses services from other subsystems to send or request the relevant data.

The WindowController also uses the Access Control subsystem to check the privileges of the current user and select between the PatientViewWindow and the SysAdminViewWindow, as well as enable or disable GUI elements as appropriate.[3]

Traceability	All use cases Appendix I (EXCLUDING UC-700 and EXTENDS use cases), UC-830, UC-840

Access Control

The Access Control subsystem provides basic application access control (i.e. security) services to the GUI subsystem. Although the AccessControl class does not actually restrict access to any other cuCare subsystems, it does provide the GUI subsystem with the current login status, indicating whether the system is currently in a state of having authorized a particular user and what level of access that user has. The Access Control subsystem uses the services of the Client Data Storage subsystem to retrieve user data from the database on the cuCare server.

Traceability	UC-100, UC-110, UC-820

[3] Physician and Administrative Assistant users share the same interface (PatientViewWindow) but have different access privileges.

System Administration

The System Administration subsystem provides a service accessible to the GUI that allows a user logged in as a System Administrator to configure different aspects of the system. This subsystem provides services for use by the GUI subsystem. The GUI can use these services to request a change to the server audit time or the IP address/port used to make a connection from the client to the server by sending the appropriate new value with the appropriate service. The SystemAdministration class receives these change requests, interprets them, and uses services provided by the Client Data Storage Subsystem to send the requests to their intended final recipient. One the request is made, the SystemAdministration class waits for confirmation, and then informs the GUI subsystem as to whether the operation succeeded or failed.

Traceability	UC-640, UC-650, UC-660, UC-670, UC-680

User Data Management

The User Data Management subsystem provides a service to the GUI subsystem for retrieval and storage of user profile data as well as the creation and deletion of users authorized to use the cuCare system. The UserData class uses the services of the Client Data Storage subsystem to retrieve user data from the database on the cuCare server.

Traceability	UC-600, UC-610, UC-620, UC-630, UC-850

Patient Data Management

The Patient Data Management subsystem provides a service to the GUI subsystem for retrieval and storage of patient health records, including physician consultations, follow-up requests, and contact information. This subsystem is also responsible for handling requests to create new patient records as well as requesting that a patient record be deleted from the system. The PatientData class subsystem uses the services of the Client Data Storage subsystem to store and retrieve patient data to and from the database on the cuCare server.

Traceability	UC-200, UC-210, UC-220, UC-230, UC-300, UC-310, UC-320, UC-330, UC-340, UC-400, UC-410, UC-420, UC-430, UC-440, UC-450, UC-460

Reports Management

The Reports Management subsystem provides a service to the GUI subsystem for generation of reports. The ReportsManagement class interprets the report request, generates the appropriate sequence of request(s) for the Client Data Storage Subsystem, and processes these requests. Once all of the necessary data has been obtained, the ReportsManagement class constructs the requested report and passes it back to the GUI subsystem to be displayed to the user.

Traceability	UC-500, UC-510, UC-520, UC-530, UC-540, UC-550, UC-560, UC-570, UC-580, UC-590, UC-595

Client Data Storage

The Client Data Storage subsystem handles allocation, access, and de-allocation of model objects while they are in use by the client.

The Factory class instantiates model objects when new ones are created in the client, or when their data is returned from the communications subsystem. All dynamic allocation of model objects happens in this class. The Factory employs the Object Interpreter class to translate model objects to and from data use for interaction with the Communications subsystem.

The Warehouse class keeps an STL map for each type of model object. The keys are the unique ids of those model objects, and the values are pointers to the objects themselves. These maps are in place to provide access to model objects by ids. The Warehouse also handles pull requests to the Communications subsystem.

The NetworkSettings class simply stores the IP and port that should be used to connect to the server. When these values change, it instantiates a new ClientNetworkInterface object (the top level object of the Communications subsystem).

The final class in Client Data Storage, called ClientData, abstracts the Factory and Warehouse functionality into a single interface. All the subsystem services are provided by the ClientData class, which acts as a facade for the rest of the subsystem. More information on the use of the facade can be found in the section on design patterns.

Traceability	All use cases Appendix I (EXCLUDING UC-700 and EXTENDS use cases)

Object Interpreter

The Object Interpreter subsystem provides services to translate a model object into a map of name-value pairs that can be used to by the Communications subsystem, and the reverse, which fills out a model object using the data from a map of name-value pairs.

The Interpreter class performs these operations without needing to know which specific type of model object it is working with. To achieve this, the abstract base class ModelObject (from which all model objects inherit) maintains a list of properties. These properties include a name (which is the name of the data member they represent), and two pointers to get and set member functions for the corresponding data member. Thus, the object interpreter can loop through the properties, using the function pointers to get or set the data members of the model object, without knowing its type. The advantage of this approach is that changes to model objects will not break serialization or deserialization into the map format required by the Communications subsystem (as long as the list of properties is also updated in the model object's constructor).

Traceability	All use cases Appendix I (EXCLUDING UC-700 and EXTENDS use cases)

Communications

The Communications subsystem is responsible for communications between the client and server components of cuCare. It provides Retreival and Storage services to the Client Data Storage subsystem.

In terms of implementation, this means that, on behalf of a client process, the Communications subsystem requests a connection with the server, and sends a message to the server. The part of the subsystem that is active on the server receives the message, properly decomposes it, and passes the relevant information along to the ServerController subsystem. This will cause the ServerController to either store some new information, or return some information to the Communications subsystem. Once the server has done its job, the Communications subsystem sends a message back to the client with the appropriate information.

Traceability	All use cases Appendix I (EXCLUDING UC-700 and EXTENDS use cases), UC-810

Server Controller

The Server Controller subsystem is responsible for interactions with the Storage subsystem. It initiates the running of audits, and provides the Request Management service to the Communications subsystem. Audits are triggered at a consistent time every day, and they cause an update of Storage that flags any follow-up appointments that have expired. The Request Management service allows messages that come from the client to initiate the storage and retrieval of data. Essentially, Server Controller acts as a general-purpose controller for server's persistent storage. In the event that more storage manipulation options are added to the server in the future, Server Controller would be the subsystem responsible for coordinating them.

Traceability	All use cases Appendix I (EXCLUDING UC-700 and EXTENDS use cases)

Storage

The Storage subsystem provides the cuCare system with access to the database, while abstracting away the use of SQL statements. The Repository class provides the interface which the Server Controller subsystem uses. Repository holds a Database type pointer. Database is an abstract class which guarantees functions for opening a connection to and querying a relational database, without regard to the specifics of the database implementation. SQLite3Database is a subclass of Database, which provides the guaranteed operations using an SQLite3 database. The QueryResult class wraps a vector of vectors of strings representing the rows and columns of the table returned by SQLite. It provides methods which simplify the access of the result data.

Currently, the Repository will always instantiate the Database pointer with an SQLite3Database object, as it is the only concrete implementation supported in this phase. The storage subsystem also handles running the audit. This is a simple operation, as the entire audit process can be completed with a single SQL query. (eg. UPDATE followups SET status = 'overdue' WHERE [date > current date] AND status = 'pending')

Traceability	All use cases Appendix I (EXCLUDING EXTENDS use cases)

2.3 Design Evolution

This section details the differences in design between the first prototype and the proposed cuCare system design. Where necessary, rationale for the selected approach is provided.

GUI Window Management

Both designs provide a GUI subsystem. The prototype contained two main window classes, LoginWindowDialog and MainWindow, which made use of two other window classes, PatientSelectDialog and FollowupTypeSelectDialog. The LoginWindowDialog and MainWindow classes were not aware of each other, which made it difficult to implement the functionality of logging out of the system and logging back in as another user.

The revised design calls for all windows to be created and destroyed by the WindowController class. This greatly simplifies implementation of window management, and provides for a more logical flow of handling user-generated events.

In addition, the revised design reduces coupling, as the WindowController class is now the only class in the GUI subsystem that uses services from other subsystems. This was not the case in the prototype, where the individual window classes were allowed direct access to services provided by other subsystems.

MasterController

The new design calls for the services provided by the old MasterController class to be distributed over several different classes, to reduce coupling and increase cohesion. As such, the new design calls for five new classes, each corresponding to a subset of use cases, grouped based on the data entities being manipulated (e.g. user accounts, patients, consultations, etc.) In addition, the responsibility for persistent data, as well as allocating and de-allocating memory as required by the cuCare client application, has been abstracted and moved to the Client Data Storage subsystem.

Security

At present the Access Control subsystem provides information to the WindowController class about what elements of the GUI are allowed to be displayed based on the current user type. As such only the most basic client-side security is provided.

Future development of the cuCare system would be well advised to include a more robust access control functionality. As it stands, the server does not implement any security features, and is as such vulnerable to unauthorized data storage and retrieval.

Client Data Storage

The Client Data Storage subsystem deals with the management of the flow of data between the GUI and the cuCare server process. It follows the façade design pattern to aggregate and simplify data storage and retrieval services, without having to reveal the details of what those services entail.

In the new implementation, the Factory class is responsible for instantiating any model objects created by the client or returned from the server. The Warehouse class stores all objects currently instantiated

in the client in STL[4] maps that are keyed by the object's unique ID. This design allows for fast retrieval of objects that are requested by the GUI, eliminating the vector traversal commonly employed in the first prototype.

Object Interpreter

The Object Interpreter is a new subsystem, in part due to the fact that the prototype system did not include a working communications component. This subsystem translates model objects to and from STL maps of key-value pairs that are passed to the Communications subsystem.

The services provided by the Object Interpreter subsystems are used by the Client Data Storage subsystem to enable the use of the Communications subsystem to send requests and data to and from the cuCare server. This approach reduces coupling between the Client Data Storage subsystem and the Communications subsystem, allowing for changes to be made to the way the client application interacts with the communications subsystem without having to make changes in the Client Data Storage subsystem itself.

Communications

There are a number of differences between the design of the first prototype and the cuCare design proposed by this document. However, the most notable change is the addition of a Communications subsystem. The communications component of the first prototype was not completed in time. In light of this, the decision was made to place the Storage subsystem and its constituent classes in the client application process, so as to enable the MasterController class to make data storage and retrieval requests.

The revised design calls for a Communications subsystem to be implemented. This will fulfill the requirements elicited from the cuCare system description, and make it possible to deploy cuCare on multiple client nodes, with a single server node providing persistent data storage and retrieval services.

The new design calls for the messages used to communicate between client and server to be serialized C++ maps with keys and values of uniform types. Pieces of these maps are used to represent objects, on the server side. The maps also contain special indicators like "object type" and "request type" that are used to determine the appropriate behavior to carry out when a message is received.

Qt provides functions that make it very easy to convert these maps to and from a QByteArray, which is a Qt object that can be sent and received by a QTcpSocket. Since this messaging protocol is so simple, a format like JSON or XML was not really required. It would have been unwise to put extra time and effort into implementing such a format, when the extra functionality it provides may never have been needed.

The Communications subsystem is heavily layered. It consists largely of classes that use only one other class in the subsystem, and who do not know what classes they themselves are being used by. This allows many of the classes to change without affecting the layer directly above them.

[4] C++ Standard Template Library

The layers on the client side are essentially a mirror of the layers on the server side. On each side of the mirror, there are three layers: a layer that understands the different pieces that make up a message, a layer that knows how to serialize and de-serialize a message, and a layer that knows how to deal with sending and receiving data. This way, any of these three mirrored layers can change some of their behavior, and the other two can often remain the same. For instance, if we were to decide to use JSON, only the layer that deals with serialization and de-serialization would need to change.

The Communications Subsystem's job of sending and receiving a message is a process that involves very little decision making and has very little state to maintain. As such, it is somewhat function-oriented in its design. That is, its operation consists largely of functions that either simply call other (static) functions, or create an object, call some function in it, and subsequently destroy it. Following this, many of the classes have few data members, and, where possible, are simply abstract classes composed of public static functions. Having abstract classes like this (especially on the server) means that there is no need for creation and destruction of objects every time a new request comes in. This means there is less waste of memory and processor time. Since the server may be dealing with numerous incoming requests, it needs to put high emphasis on efficiency. The overall goal behind the design is to keep a simple and fast Communications Subsystem that is still modular and extensible.

Server Controller

The new subsystem called the Server Controller has been situated between the Communications and Storage subsystems. This class manages signals from the auditTimer, ServerNetworkInterpreter, Repository classes prompting the required operations to begin or execute a data storage or retrieval operation.

The introduction of the Server Controller potentially lays the foundation for a multi-threaded server application. However, a multi-threaded implementation is not being explored at this time.

Repository

The other reason for the addition of the Server Controller subsystem is due to major changes to the interface and data storage methods of the Storage subsystem, and more specifically the Repository class. For Prototype #1 implementation, different methods were provided for pulling, pushing, and creating each type of object that could be stored on the server.

The new implementation replaces all of those type-specific methods with three generic pull, push and create functions. These functions are designed to require only a table name string and a map of name-value pairs, which are used construct an SQL statement. The addition of the table name allows this to happen correctly without Storage knowing exactly which table is being modified, and therefore storage does not need separate push, pull, and create methods for each type of object stored. This change increases the readability and maintainability of the Repository, and removes the extremely tight coupling to the Model objects which existed in the first prototype.

The downside of this approach is a loss of some type safety. Namely, if Repository is provided with completely incorrect data, it will not be able to distinguish this from normal data in any meaningful way.

Conclusions

Most notably, the new system design enables inter-process communications, making a client-server implementation possible. It is also worth noting that through the use of established design patterns coupling between most of the subsystems has been significantly reduced. Other changes have also been introduced to improve efficiency or solve other minor problems. As such, the current design calls for a both more complete and more modular implementation of the cuCare system as compared to the design of Prototype #1.

3 Design Strategies

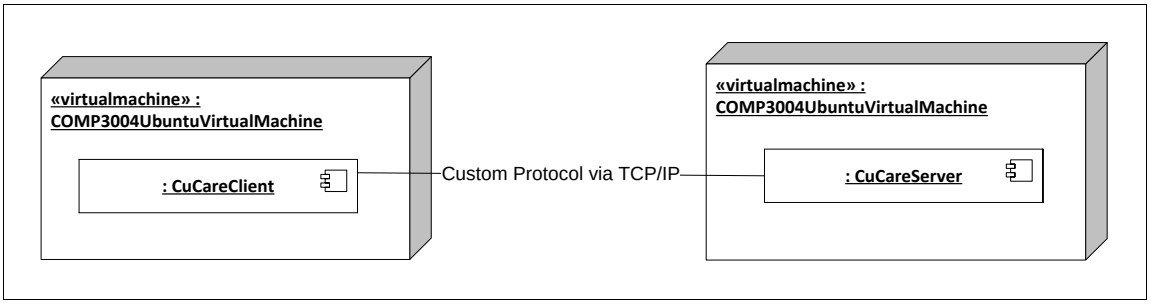
This section presents the hardware to software mapping of system components, a summary of the strategy employed to provide persistent data management, as well as an overview of how design patterns are employed in the proposed cuCare system design.

3.1 Hardware / Software Mapping

Components and Nodes

There are two major run-time components that make up the cuCare system: a client component and a server component, each residing on separate nodes. The proposed client-server architecture calls for the two components to communicate with each-other over a network, making it possible to provide multiple client nodes real-time access to a central server. Providing cuCare's services to multiple clients in different locations is a key requirement of the cuCare system.

Figure 5 - cuCare - Components to Nodes Mapping



Note on the Database

Though we are using a SQLite database for storage in our implementation of cuCare, it does not make up its own component. This is because an SQLite database does not run in a separate process. Instead, the database file itself is accessed through library functions provided by the Sqlite3 C++ API (which are used by the Storage subsystem). We make the assumption that a component is defined as having its own process, and our SQLite database does not meet this criterion.

Subsystem Mappings

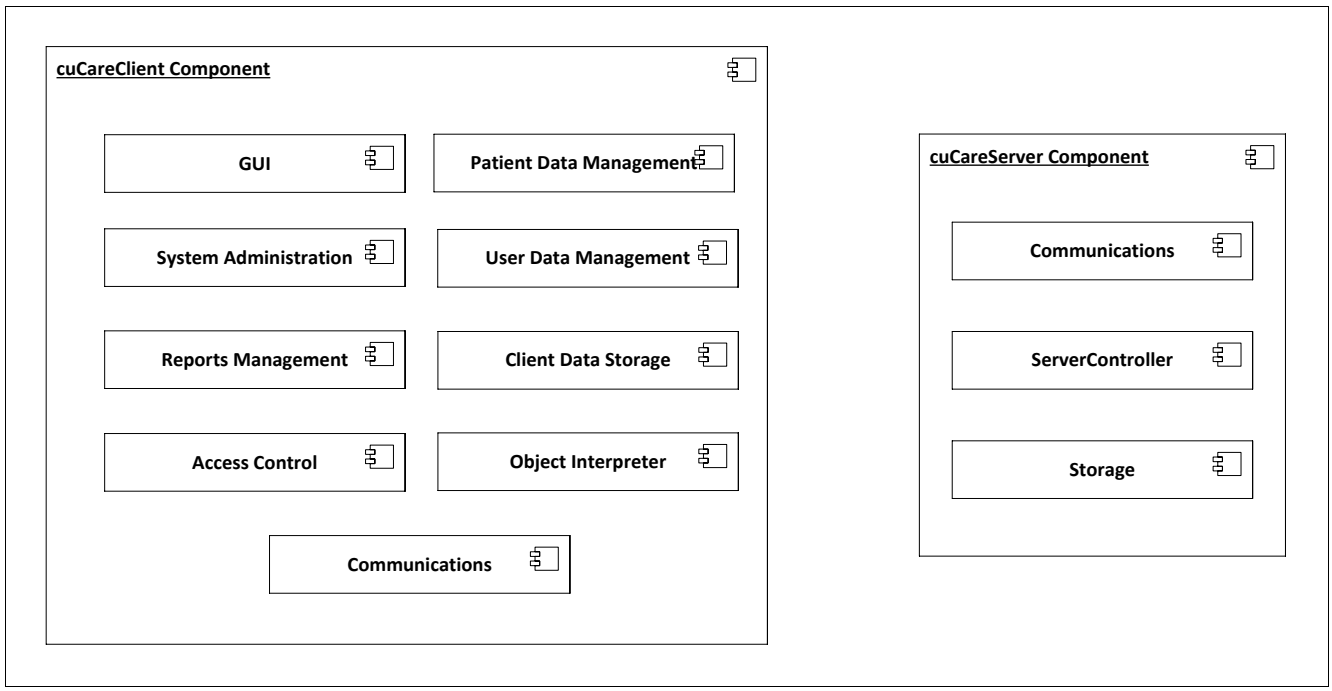
All of the subsystems pertaining directly to application logic, user interaction, and the object model (all of the subsystems above the communications subsystem) belong to the client component. This places a lot of the code in the client. However, this is not a bad thing. Since cuCare only supports the use of a single server process, which may be relied upon by numerous clients, server processing time could become a scarce commodity. Keeping as much of the logic and memory usage as reasonably possible on the client side rather the server side will reduce server lag during times of congestion.

The communications subsystem is implemented as a library, and is contained within both the client and server components, as both will need to communicate over their networks. Since the Communications subsystem is used by both components, instead of separate communications subsystems for each, any

changes made to the method of communication will only need to be implemented in one library. In addition, due to the layered structure of the communications subsystem, fundamental changes can be made to the inner layers without affecting any other pieces of the client and server components. In addition, keeping most of the logic in one place eliminates the possibility of inconsistencies cropping up between two components that are attempting to follow the same rules. This contributes to the overall reliability of the system.

The remaining subsystems pertain to dealing with persistent storage, and so they belong to the server component of cuCare. The server subsystems are focused almost completely on accessing and updating storage. Due to the fact that most of the logic is contained within the client component, the server is free to direct its processing time toward the core of its purpose – executing queries. This design allows the server component to be as light and efficient as possible.

Figure 6 - cuCare - Subsystems to Components Mapping



Note: the presence of the assignment of all subsystems to components and of all components to nodes constitutes an assignment of all subsystems to nodes.

3.2 Persistent Data Management

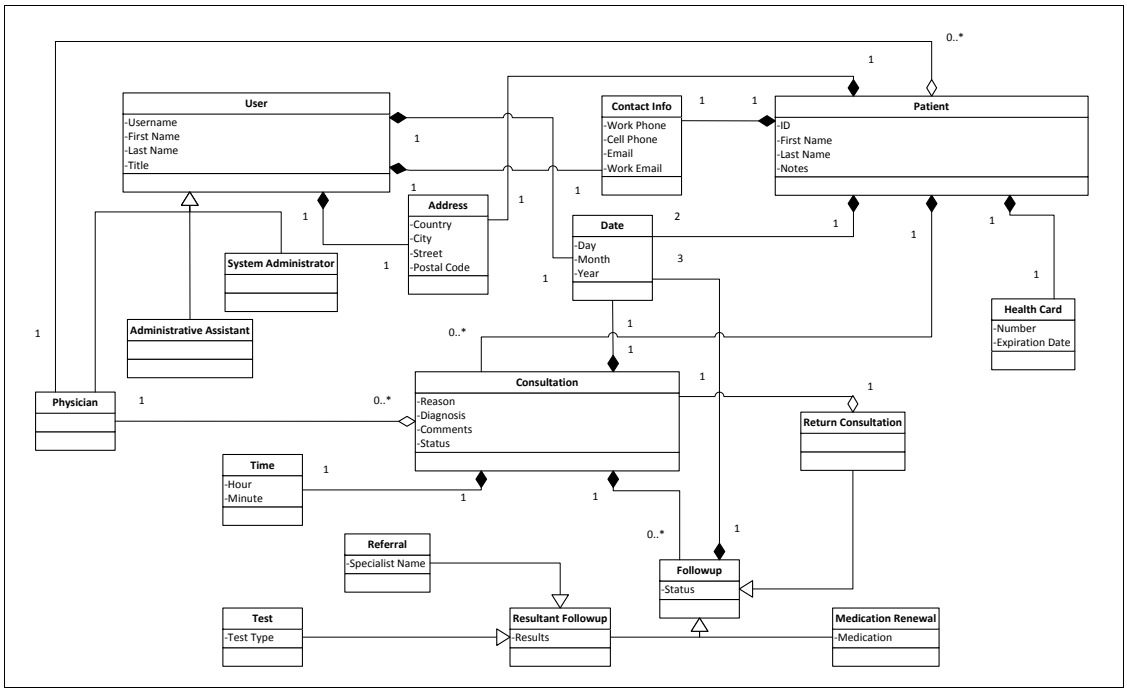
The system design proposed herein calls for persistent data storage to be implemented via a relational database. The specific SQL implementation used in both Prototype #1 and the system design proposed in this document is SQLite v3. SQLite provides a C++ interface library and does not require an external database server, making it an optimal choice given the scope of the project. It is worth noting that as per the discussion in the Design Evolution section the proposed subsystem decomposition abstracts database functionality in a manner that makes it easy to switch the specific database implementation should this be desired.

There are several reasons why a relational database was chosen over flat file storage.

- First, using a relational database with a well designed schema avoids data duplication. Because a record never has to be changed in multiple places, the possibility of records becoming inconsistent is avoided.
- Second, a relational database is more reliable in avoiding data loss in the event of a failure. Should the program crashes while a flat file is open and being rewritten, all the data in that file could be lost. With a relational database, a crash will not wipe the database – at most all that will be lost is the data manipulated by the transaction at the time of failure.
- Finally, a relational database allows us to perform complex queries quickly and efficiently using SQL statements, which is not a feature of flat files.

Most of the persistent data present in the cuCare system is captured by the cuCare object model. The object model is implemented via the classes within the Model library.

Figure 7 - cuCare - Object Model



Data Mapping

Below are the steps which were followed to map the object model on to the table schema:

1. Where an object has a 1-to-1 (or 1-to-2 or 3 in a few cases) composition (black diamond) relationship, the fields of the smaller object which the larger object is composed of are folded into the larger object.
2. A unique id integer and 'deleted' boolean are added to each remaining object. In the case of inheritance hierarchies, the fields are added in the base class and inherited in the subclasses. The unique ids provide the primary keys for the database tables, while the deleted booleans let us mark objects that have been removed, without potentially causing dependence problems by actually removing the data.
3. Where an object has a one-to-many aggregation (white or black diamond) relationship, the object on the 'many' side gains a field indicating the id of the object on the 'one' side of the relationship.
4. For each base class object, a database table is created with one column for each field. The primary key of each table is the unique id of that object. Fields which contain the unique ids of other types of object become foreign keys referencing the table for that object. All fields are of type TEXT unless referring to unique id. Fields referring to unique ids are of type INT.
5. For each subclass object, a database table is created with one column for each field that was added in that subclass, as well as a field for the unique id that was inherited from the base class. The unique id becomes both the primary key and a foreign key to the base class table. The same strategy of unique ids of other object types becoming foreign keys as in the previous step is applied. Again, all fields are of type TEXT unless referring to unique id (in which case they are of type INT).
6. The username field has a unique constraint applied to it, because users use the username instead of the unique database id to log in.

The resulting entity relationship (ER) diagram and the final table schema are presented on the next page.

Figure 8 - cuCare - Persistent Data Management - ER Diagram

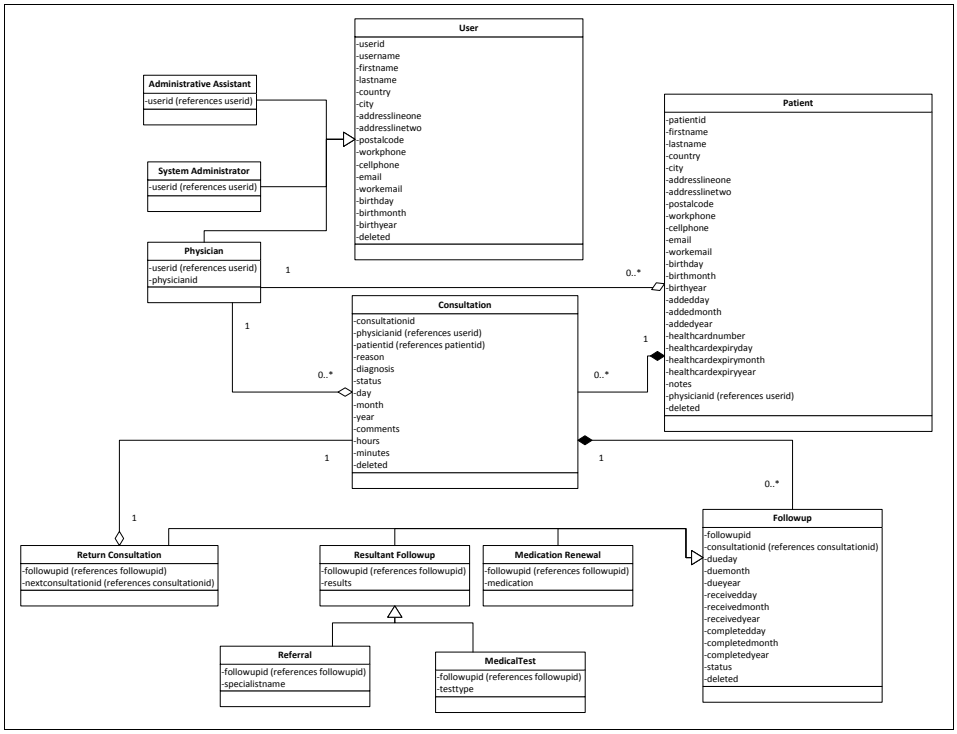


Figure 9 - cuCare - Persistent Data Management - Table Schema

Table Name	*Field Names*
USERS	userid, username (unique), firstname, lastname, country, city, addresslineone, addresslinetwo, postalcode, workphone, cellphone, email, workemail, birthday, birthmonth, birthyear, deleted
PHYSICIANS	userid (foreign key USERS.userid), physicianid (unique)
ADMINASSISTANTS	userid (foreign key USERS.userid)
SYSADMINS	userid (foreign key USERS.userid)
PATIENTS	patientid, firstname, lastname, country, city, addresslineone, addresslinetwo, postalcode, workphone, cellphone, email, workemail, birthday, birthmonth, birthyear, addedday, addedmonth, addedyear, healthcardnumber, healthcardexpiryday, healthcardexpirymonth, healthcardexpriryyear, notes, physicianid (foreign key PHYSICIANS.userid), deleted
CONSULTATIONS	consultationid, physicianid (foreign key PHYSICIANS.userid), patientid (foreign key PATIENTS.patientid), reason, diagnosis, status, day, month, year, comments, hours, minutes, deleted
FOLLOWUPS	followupid, consultationid (foreign key CONSULTATIONS.consultationid), dueday, duemonth, dueyear, receivedday, receivedmonth, receivedyear, completedday, completedmonth, completedyear, status, deleted
RESULTANTFOLLOWUPS	followupid (foreign key FOLLOWUPS.followupid), results
MEDICATIONRENEWALS	followupid (foreign key FOLLOWUPS.followupid), medication
RETURNCONSULTATIONS	followupid (foreign key FOLLOWUPS.followupid), nextconsultationid (foreign key CONSULTATIONS.consultationid)
REFERRALS	followupid (foreign key RESULTANTFOLLOWUPS.followupid), specialistname
MEDICALTESTS	followupid (foreign key RESULTANTFOLLOWUPS.followupid), testtype

3.3 Design Patterns

Software design patterns provide a means of resolving common design challenges while maintaining high cohesion within the subsystem and low coupling with other subsystems. This section cites the instances where the proposed cuCare system design makes use of established software design patterns and provides the rationale for the selected approach.

Bridge

The bridge pattern is used when you have a component in your system that you wish to be able to modify or possibly swap out without knowing much about the rest of the system. The implementation of the component is very loosely coupled to its abstraction, which makes this possible.

CuCare employs the bridge design pattern in the Storage subsystem. The abstraction in this case is the Repository class, which is responsible for storing data for the system but is unaware of the actual method of storage. The Sqlite3Database class is the implementor, inheriting from the Database class to interpret storage requests and store them in a database. Should another storage method be required, the system modification could be accomplished by simply replacing the Database class with a new class that provides the desired storage implementation.

Command

The command pattern is used to encapsulate all of the parts of a method call into an object, so that it can be called at a later time. This pattern is employed in our system design in the handling of client-server communications concerning data retrieval and storage. More specifically, the user's input and data retrieval requests are encapsulated by the client's Communications subsystem and sent over the network. The request and all of its parameters are then de-seralized by the server's Communications subsystem and passed on through the Server Controller subsystem, to be executed within the Storage subsystem.

Façade Pattern

The façade pattern is used to simplify an interface to a larger or more complex interface or interfaces. cuCare uses this design pattern to simplify the interface to the Client Data Storage Subsystem. The ClientData class acts as the façade for this subsystem, providing a way for other subsystems (Access Control, Patient Data Management etc.) to request data or submit changes to existing data, without having to know which specific class in the subsystem actually handles the request. This decreases the coupling between the Client Data Storage subsystem and other subsystems that use its services.

Singleton Pattern

The singleton pattern restricts an object to one instantiation. Generally this single instantiation can be returned from a static function of the singleton class. This allows the class to exist and be available for use without being explicitly instantiated by any other subsystem.

The singleton classes used by our system are ClientData and ServerController, located in the Client Data Storage subsystem, and Server Controller subsystem respectively.

In the case of the ClientData class, it is necessary that the same instance be available with equal priority to several outside subsystems neither of which is aware of the others. We decided the best way to handle the situation was to provide static access to a singleton object. This way no other subsystem is responsible for the explicit creation of the ClientData object, and all subsystems are guaranteed to access the same instance.

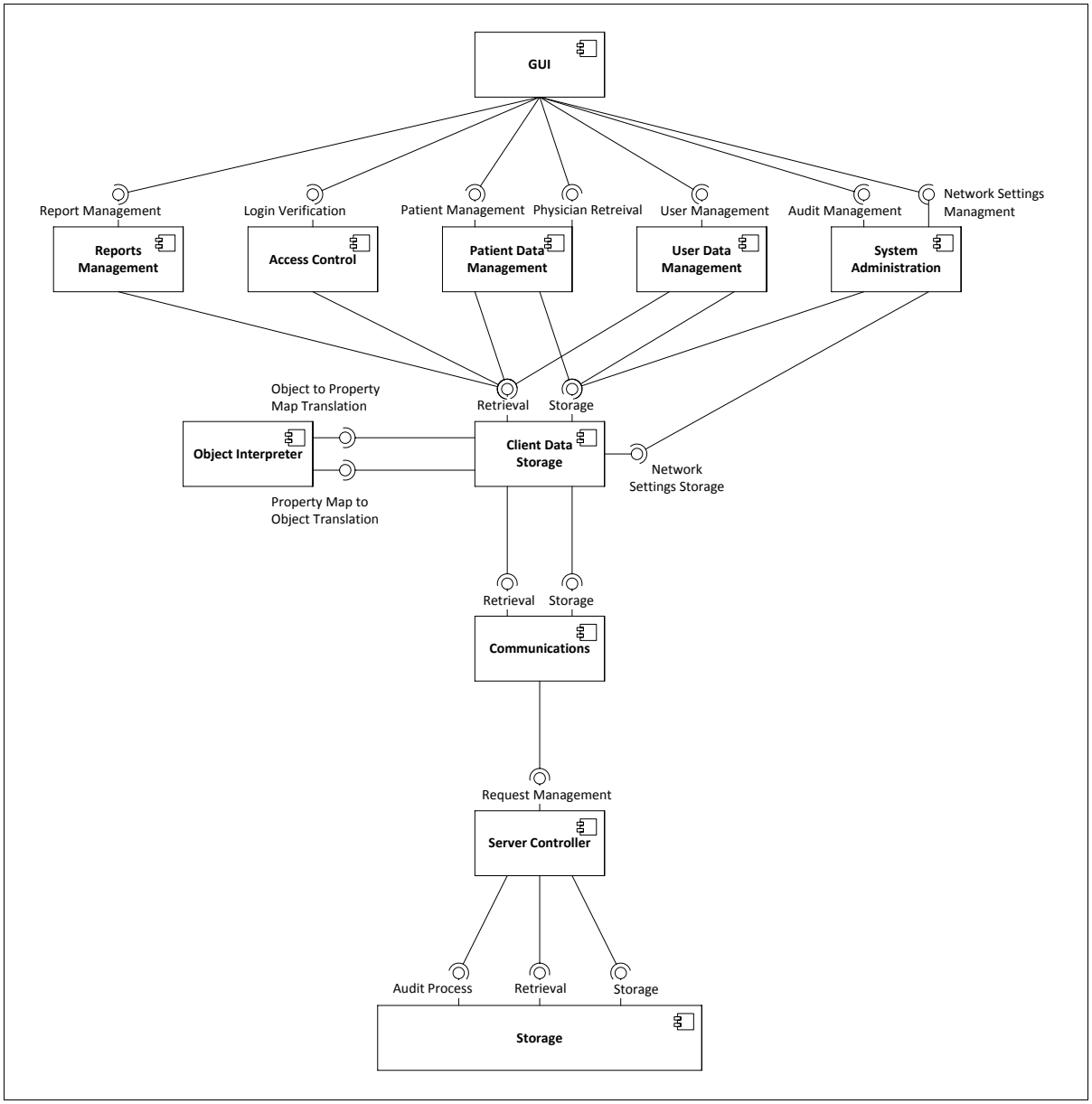
In the case of the ServerController, the main motivation for using a singleton pattern is that it eliminates a large amount of redundant parameter passing in the Communications subsystem. Due to the structure of the Communications subsystem, without the singleton pattern, the ServerNetworkListener class in Communications would need to maintain a pointer to a ServerController that would need to be passed through numerous levels of function calls every time a request was sent to the server. A secondary reason for using the singleton pattern here is that, given the design of our system, it is very likely that there will never need to be more than one instance of ServerController.

4 Subsystem Services

This section provides a description of the services offered by each subsystem of cuCare. For each service a description of its operations is provided, specifying the class to which each operation belongs. The entire set of system services is depicted by a component diagram using the UML ball-and-socket notation (below).

NOTE: Operation description tables include a pseudo code prototype to provide additional information about the operation. Actual function declarations are found in the Class Interfaces section.

Figure 10 - cuCare - Subsystem Services



System Administration

Network Settings Management

The Network Settings Management service provides the GUI with a means of viewing and modifying the cuCare client application network settings, including the IP and port number of the cuCare server.

Subsystem	System Administration
Class	SystemAdministration
Service	Network Settings Management
Operation	Set Client IP Address
Prototype	*(void) SystemAdministration::Set Client IP Address (string ipAddress)*
Description	This operation allows the GUI to inform this subsystem that the user has changed the IP address on which the client should connect to the server. This data is then sent to Client Data Storage using the Network Settings Storage service.
Traceability	UC-660

Subsystem	System Administration
Class	SystemAdministration
Service	Network Settings Management
Operation	Set Client Port Number
Prototype	*(void) SystemAdministration::Set Client Port Number (int portNumber)*
Description	This operation allows the GUI to inform this subsystem that the user has changed the port number on which the client should connect to the server. This data is then sent to Client Data Storage using the Network Settings Storage service.
Traceability	UC-670

Audit Management

The audit management service allows the GUI to view and modify the current time at which the audit is run.

Subsystem	System Administration
Class	SystemAdministration
Service	Audit Management
Operation	Set Audit Time
Prototype	*(void) SystemAdministration::Set Audit Time (Time time)*
Description	This operation allows the GUI to inform this subsystem that the user has changed the time at which the automatic audit should take place. This data is sent to to the Client Data Storage using the Storage service.
Traceability	UC-650

Subsystem	System Administration
Class	SystemAdministration
Service	Audit Management
Operation	Get Audit Time

Prototype	*(Time t) SystemAdministration::Get Audit Time ()*
Description	This operation allows the GUI to request the current audit time so that it can be displayed to the user. This data is obtained using the Retrieval service of the Client Data Storage subsystem.
Traceability	UC-650

Reports Management

Report Management

This service allows the GUI to set up a report, including choosing report type, adding filters, and entering other information. It then allows you to generate that report, and save it.

Subsystem	Reports Management
Class	ReportsManagement
Service	Report Management
Operation	Set Report Type
Prototype	*(void) ReportsManagement::Set Report Type (ReportType reportType)*
Description	This operation allows the GUI to inform this subsystem of the type of report that the user wishes to run. The information is stored in this subsystem and used during the *Generate Report* operation.
Traceability	UC-500, UC-510, UC-520, UC-530, UC-540, UC-550, UC-560

Subsystem	Reports Management
Class	ReportsManagement
Service	Report Management
Operation	Create, Modify, or Remove Filter
Prototype	*(void) ReportsManagement::Create, Modify, or Remove Filter (Filter filter)*
Description	This operation allows the GUI to inform this subsystem that the user has added, changed, or removed a report filter. The information is stored in this subsystem and used during the *Generate Report* operation.
Traceability	UC-570, UC-580

Subsystem	Reports Management
Class	ReportsManagement
Service	Report Management
Operation	Set Report Parameters
Prototype	*(void) ReportsManagement::Set Report Parameters (Varies by report type)*
Description	For reports which require additional information, this operation allows the GUI to inform this subsystem of the additional information the user has entered. The information is stored in this subsystem and used during the *Generate Report* operation.
Traceability	UC-500, UC-510, UC-520, UC-530, UC-540, UC-550, UC-560

Subsystem	Reports Management

Class	ReportsManagement
Service	Report Management
Operation	Generate Report
Prototype	*(Varies by report type) ReportsManagement::Generate Report ()*
Description	This operation allows the GUI to inform this subsystem that the user requested the report to be generated. Using the information collected during other operations, this subsystem calls the *Get Report Results* operation in Client Data Storage. When results are returned, this operation then returns appropriate data to the GUI.
Traceability	UC-590

Subsystem	Reports Management
Class	ReportsManagement
Service	Report Management
Operation	Save Report
Prototype	*(void) ReportsManagement::Save Report (filestream file)*
Description	This operation allows the GUI to inform this subsystem that the user has requested the generated report be saved. If this operation is called before a report has been generated, an exception is thrown. If a report has been generated, this subsystem outputs that data as a text file in the location provided.
Traceability	UC-595

Access Control

Login Verification

This service allows the GUI to have a username checked for status as a valid physician, administrative assistant, or system administrator.

Subsystem	Access Control
Class	AccessControl
Service	Login Verification
Operation	Verify Username
Prototype	*(LoginStatus) AccessControl::Verify Username (string username)*
Description	This operation allows the GUI to request verification of a username entered by the user. This subsystem requests all users of each type from Client Data Storage, and then compares the username against these entries, attempting to find a match. If a match is found a login status with the type of user is returned. If not, the failed valued for login status is returned. We are aware that our current approach for login verification is not a good approach. In future design iterations we would like to handle verification on the server, both for security and efficiency reasons.
Traceability	UC-100, UC-110

Subsystem	Access Control
Class	AccessControl
Service	Login Verification

Operation	Login Status
Prototype	*(LoginStatus) AccessControl::Get Login Status ()*
Description	This operation allows the GUI to request the current login status.
Traceability	UC-100, UC-110

Patient Data Management

Patient Management

This service allows retrieval, creation, and modification of patients and their nested objects, consultations and follow-ups.

Subsystem	Patient Data Management
Class	PatientData
Service	Patient Management
Operation	Get Patient List
Prototype	*(List<Patient>) PatientData::Get Patient List (FilterType filterType, Physician physician, FollowupType followupType)*
Description	This operation allows the GUI to request a list of all patients. The related consultations and followups are not retrieved. The patient objects only store the ids of their consultations. The filter type specifies whether the patient list should be filtered by physician and/or followup status. The physician and followup type are only used if those filters are turned on. Otherwise they are ignored. This subsystem uses the Retrieval service of the Client Data Storage to get the list of patients, then returns it to the GUI for display.
Traceability	UC-210

Subsystem	Patient Data Management
Class	PatientData
Service	Patient Management
Operation	Get Full Patient
Prototype	*(Patient) PatientData::Get Full Patient (Patient patient)*
Description	This operation allows the GUI to request all data for a specific patient. The consultations associated with that patient are pulled from the server, as are the followups associated with those consultations. The pulled information remains in the Client Data Storage warehouse until the next time the operation is called. This subsystem uses the Retrieval service of the Client Data Storage to pull the patient data.
Traceability	UC-210, UC-310, UC-410

Subsystem	Patient Data Management
Class	PatientData
Service	Patient Management
Operation	Create Patient
Prototype	*(void) PatientData::Create Patient (Patient patient)*
Description	This operation allows the GUI to inform this subsystem that the user has created a new

	patient. The patient object that this operation takes does not have an id yet, and is not dynamically allocated. It is used to contain the data needed to send a create request to the server. This subsystem uses the Storage service of the Client Data Storage to create the patient. After that is completed, the patient is available from the Client Data Storage warehouse.
Traceability	UC-200

Subsystem	Patient Data Management
Class	PatientData
Service	Patient Management
Operation	Modify Patient
Prototype	*(void) PatientData::Modify Patient (Patient patient)*
Description	This operation allows the GUI to inform this subsystem that the user has modified a patient. The patient object that this operation takes is not dynamically allocated. It is used to contain the data needed to send a push request to the server. This subsystem uses the Storage service of the Client Data Storage to push the patient changes. After that is completed, the new version of the patient is available from the Client Data Storage warehouse.
Traceability	UC-220, UC-230

Subsystem	Patient Data Management
Class	PatientData
Service	Patient Management
Operation	Create Consultation
Prototype	*(void) PatientData::Create Consultation(Consultation consult)*
Description	This operation allows the GUI to inform this subsystem that the user has created a new consultation. The consultation object that this operation takes does not have an id yet, and is not dynamically allocated. It is used to contain the data needed to send a create request to the server. This subsystem uses the Storage service of the Client Data Storage to create the consultation. After that is completed, the consultation is available from the Client Data Storage warehouse.
Traceability	UC-300

Subsystem	Patient Data Management
Class	PatientData
Service	Patient Management
Operation	Modify Consultation
Prototype	*(void) PatientData::Modify Consultation(Consultation consult)*
Description	This operation allows the GUI to inform this subsystem that the user has modified a consultation . The consultation object that this operation takes is not dynamically allocated. It is used to contain the data needed to send a push request to the server. This subsystem uses the Storage service of the Client Data Storage to push the consultation changes. After that is completed, the new version of the consultation is

	available from the Client Data Storage warehouse.
Traceability	UC-320, UC-330, UC-340

Subsystem	Patient Data Management
Class	PatientData
Service	Patient Management
Operation	Create Followup
Prototype	*(void) PatientData::Create Followup(Followup followup)*
Description	This operation allows the GUI to inform this subsystem that the user has created a new followup. The followup object that this operation takes does not have an id yet, and is not dynamically allocated. It is used to contain the data needed to send a create request to the server. This subsystem uses the Storage service of the Client Data Storage to create the followup. After that is completed, the followup is available from the Client Data Storage warehouse.
Traceability	UC-400

Subsystem	Patient Data Management
Class	PatientData
Service	Patient Management
Operation	Modify Followup
Prototype	*(void) PatientData::Modify Followup(Followup followup)*
Description	This operation allows the GUI to inform this subsystem that the user has modified a followup. The followup object that this operation takes is not dynamically allocated. It is used to contain the data needed to send a push request to the server. This subsystem uses the Storage service of the Client Data Storage to push the followup changes. After that is completed, the new version of the followup is available from the Client Data Storage warehouse.
Traceability	UC-420, UC-430, UC-440, UC-450, UC-460

Physician Retrieval

This service allows retrieval of a list of all physicians. This is important to the Patient Data Management subsystem, as consultations and patients each have an associated physician, which must be selected from the list.

Subsystem	Patient Data Management
Class	PatientData
Service	Patient Management
Operation	Get Physician List
Prototype	*(List<Physician>) PatientData::Get Physician List ()*
Description	This operation allows the GUI to request a list of physicians. This subsystem uses the Retrieval service of the Client Data Storage to pull all physicians, and returns this list to the GUI.
Traceability	UC-800

User Data Management

User Management

This service allows retrieval, creation and modification of the three types of user - physician, administrative assistant, and system administrator.

Subsystem	User Data Management
Class	UserData
Service	User Management
Operation	Create Physician
Prototype	*(void) UserData::Create Physician (Physician physician)*
Description	This operation allows the GUI to inform this subsystem that the user has created a new physician. The physician object that this operation takes does not have an id yet, and is not dynamically allocated. It is used to contain the data needed to send a create request to the server. This subsystem uses the Storage service of the Client Data Storage to create the physician. After that is completed, the patient is available from the Client Data Storage warehouse.
Traceability	UC-600, UC-610

Subsystem	User Data Management
Class	UserData
Service	User Management
Operation	Modify Physician
Prototype	*(void) UserData::Modify Physician (Physician physician)*
Description	This operation allows the GUI to inform this subsystem that the user has modified a physician . The physician object that this operation takes is not dynamically allocated. It is used to contain the data needed to send a push request to the server. This subsystem uses the Storage service of the Client Data Storage to push the physician changes. After that is completed, the new version of the physician is available from the Client Data Storage warehouse.
Traceability	UC-620, UC-630

Subsystem	User Data Management
Class	UserData
Service	User Management
Operation	Create Admin Assistant
Prototype	*(void) UserData::Create Admin Assistant(AdminAssistant adminAssistant)*
Description	This operation allows the GUI to inform this subsystem that the user has created a new admin assistant. The admin assistant object that this operation takes does not have an id yet, and is not dynamically allocated. It is used to contain the data needed to send a create request to the server. This subsystem uses the Storage service of the Client Data Storage to create the admin assistant. After that is completed, the consultation is available from the Client Data Storage warehouse.
Traceability	UC-600, UC-610

Subsystem	User Data Management
Class	UserData
Service	User Management
Operation	Modify Admin Assistant
Prototype	*(void) UserData::Modify Admin Assistant(AdminAssistant adminAssistant)*
Description	This operation allows the GUI to inform this subsystem that the user has modified a admin assistant. The admin assistant object that this operation takes is not dynamically allocated. It is used to contain the data needed to send a push request to the server. This subsystem uses the Storage service of the Client Data Storage to push the admin assistant changes. After that is completed, the new version of the admin assistant is available from the Client Data Storage warehouse.
Traceability	UC-620, UC-630

Subsystem	User Data Management
Class	UserData
Service	User Management
Operation	Create Sys Admin
Prototype	*(void) UserData::Create Sys Admin(SysAdmin sysAdmin)*
Description	This operation allows the GUI to inform this subsystem that the user has created a new sys admin. The sys admin object that this operation takes does not have an id yet, and is not dynamically allocated. It is used to contain the data needed to send a create request to the server. This subsystem uses the Storage service of the Client Data Storage to create the sys admin. After that is completed, the sys admin is available from the Client Data Storage warehouse.
Traceability	UC-600, UC-610

Subsystem	User Data Management
Class	UserData
Service	User Management
Operation	Modify Sys Admin
Prototype	*(void) UserData::Modify Sys Admin(SysAdmin sysAdmin)*
Description	This operation allows the GUI to inform this subsystem that the user has modified a sys admin. The followup object that this operation takes is not dynamically allocated. It is used to contain the data needed to send a push request to the server. This subsystem uses the Storage service of the Client Data Storage to push the sys admin changes. After that is completed, the new version of the sys admin is available from the Client Data Storage warehouse.
Traceability	UC-620, UC-630

Client Data Storage

Retrieval

This service deals with making pull requests to the Communications subsystem to retrieve objects from the database and store them in the warehouse, and with retrieve those objects from the warehouse for use.

Subsystem	Client Data Storage
Class	ClientData
Service	Retrieval
Operation	Get Model Object
Prototype	*(T) ClientData::Get Model Object<T> (int id) where T is a Model Object*
Description	This operation allows other subsystems to get a specific model object by it's unique id. If the object requested is not already stored in the warehouse, it will be retrieved from the server using the Communication's Retrieval service. If the object is already stored, then it is simply returned from the warehouse.
Traceability	All use cases between (and including) UC-100 and UC-460

Subsystem	Client Data Storage
Class	ClientData
Service	Retrieval
Operation	Pull Model Objects
Prototype	*(List<int>) ClientData::Pull Model Objects (string className, Map <string, string> filter)*
Description	This operation refills the warehouse with the objects returned from communications when an SQL query with the specified parameters is run on the database. This operation always uses the Retrieval service of the Communications subsystem to pull the objects, and replaces previous copies of the same object in the warehouse. The first parameter is the name of the class, which any model object class can return a constant value for. The map of strings is a partial list of the model object's properties. Those that are present in the map will become filter parameters of the SQLite query on the server. This operation returns a list of all ids of the objects which were pulled. These ids can be used to access the pulled objects in the warehouse using the *Get Model Object* operation.
Traceability	All use cases between (and including) UC-100 and UC-460

Subsystem	Client Data Storage
Class	ClientData
Service	Retrieval
Operation	Get Report Results
Prototype	*(Varies by report type) ClientData::Get Report Results (Varies by report type)*
Description	This entry represents a set of related operations (one for each type of report). This subsystem will create an appropriate pull request for the Retrieval service of the Communications subsystem, then return the results of that pull to the caller.
Traceability	UC-590

Subsystem	Client Data Storage
Class	ClientData
Service	Retrieval
Operation	Get Audit Time
Prototype	*(Time time) ClientData::Get Audit Time ()*
Description	This operation requests the current audit time from the Retrieval service of the Communications subsystem and then returns it.
Traceability	UC-650

Storage

This service deals with sending data through the Communications subsystem to be stored in the database on the server.

Subsystem	Client Data Storage
Class	ClientData
Service	Storage
Operation	Create Model Object
Prototype	*(void) ClientData::Create Model Object (ModelObject modelObject)*
Description	This operation takes a pointer to the abstract base class ModelObject which all objects in the model inherit from. It then uses the Object to Property Map Translation service of the Object Interpreter to get a map of the model object's properties. This map, along with a string indicating the model object's type (retrieved through virtual functions) is sent to the Storage service in the Communications subsystem. The unique id of the new object is set by the database on creation, and this id is passed back up through Communications to this subsystem. The newly created object is now dynamically instantiated on the client and stored in the warehouse, with the correct id.
Traceability	All use cases between (and including) UC-100 and UC-460

Subsystem	Client Data Storage
Class	ClientData
Service	Storage
Operation	Modify Model Object
Prototype	*(void) ClientData::Modify Model Object (ModelObject modelObject)*
Description	This operation takes a pointer to the abstract base class ModelObject which all objects in the model inherit from. It then uses the Object to Property Map Translation service of the Object Interpreter to get a map of the model object's properties. This map, along with a string indicating the model object's type (retrieved through virtual functions) is sent to the Storage service in the Communications subsystem.
Traceability	All use cases between (and including) UC-100 and UC-460

Subsystem	Client Data Storage
Class	ClientData
Service	Storage
Operation	Set Audit Time

Prototype	*(void) ClientData::Set Audit Time (Time time)*
Description	This operation uses the Storage service of the Communications subsystem to send a time to set the audit to run at to the server.
Traceability	UC-650

Network Settings Storage

This service allows the port and IP numbers used by Communications to connect to the server to be changed.

Subsystem	Client Data Storage
Class	ClientData
Service	Network Settings Storage
Operation	Set IP Address
Prototype	*(void) ClientData::Set IP Address (string ip)*
Description	This operation allows another subsystem to inform the Client Data Storage what ip to use when using the Communications subsystem services to connect to the server. This is stored and used whenever accessing the services of the Communications subsystem. It is also recorded in a settings file which is read on startup, so that it does not need to be set every time the program is run.
Traceability	UC-660

Subsystem	Client Data Storage
Class	ClientData
Service	Network Settings Storage
Operation	Set Port Number
Prototype	*(void) ClientData::Set Port Number (int port)*
Description	This operation allows another subsystem to inform the Client Data Storage what port number to use when using the Communications subsystem services to connect to the server. This data is stored and used whenever accessing the services of the Communications subsystem. It is also recorded in a settings file which is read on startup, so that it does not need to be set every time the program is run.
Traceability	UC-670

Object Interpreter

Object to Property Map Translation

This service translates a model object into a map of name/value string pairs based on the object's property list.

Subsystem	Object Interpreter
Class	Interpreter
Service	Object to Property Map Translation
Operation	Get Property Map
Prototype	*(Map<string, string> properties) ObjectInterpreter::Get Property Map (ModelObject*

	modelObject)
Description	This operation takes a pointer to the abstract base class ModelObject which all objects in the model inherit from. ModelObject contains a list of properties. Properties are objects with a name and two pointer to member functions to get and set a data member. The list of properties is automatically set up on creation of any model object, and allows the Object Interpreter to convert a generic ModelObject pointer into a map of property names to associated data member values without checking the type of the object. This operation then returns that map.
Traceability	All use cases between (and including) UC-100 and UC-460

Property Map to Object Translation

This service fills out the fields of a model object with the values from a map of name/value string pairs, using the model object's property list as a guide.

Subsystem	Object Interpreter
Class	Interpreter
Service	Property Map to Object Translation
Operation	Get Model Object
Prototype	*(void) ObjectInterpreter::Get Model Object (map<string, string>, ModelObject modelObject)*
Description	This operation takes a pointer to the abstract base class ModelObject which all objects in the model inherit from, and a map of properties to apply to that object. Using the modelObject's property list, properties are matched to values in the map, and set using the associate function pointer. This way the object's data can be filled out without checking the type.
Traceability	All use cases between (and including) UC-100 and UC-460

Communications

Retrieval

This service allows data to be requested from the server and returned to the caller.

Subsystem	Communications
Class	ClientNetworkInterface
Service	Retrieval
Operation	Pull
Prototype	*(List<map<string, string>>) ClientNetworkInterface::Pull (string objectType, map<string, string>)*
Description	This operation takes a map of properties to filter by and a string which will later be used to indicate which database table to look in. The data is passed from client to server, and then used for the Request Management service of the Server Controller. Once the data is retrieved, it is returned as a list of entries, each of which is a map of properties.
Traceability	All use cases Appendix I (EXCLUDING UC-700 and EXTENDS use cases)

Subsystem	Communications
Class	ClientNetworkInterface

Service	Retrieval
Operation	Get Audit Time
Prototype	*(Time time) ClientNetworkInterface::Get Audit Time ()*
Description	This operation passes a request to the server for the current audit time. The data is retrieved using the Request Management service of the Server Controller subsystem, and then returned.
Traceability	UC-650

Storage

This service allows data to be sent to the server to be stored.

Subsystem	Communications
Class	ClientNetworkInterface
Service	Storage
Operation	Create
Prototype	*(int unique id) ClientNetworkInterface::Create (string objectType, map<string, string>)*
Description	This operation takes a map of the properties of the object being created and a string which will later be used to indicate which database table to add to. The data is passed from client to server, and then used for the Request Management service of the Server Controller. The database selects a unique id for the new object, and this id is passed back up to Communications, which returns it.
Traceability	All use cases Appendix I (EXCLUDING UC-700 and EXTENDS use cases)

Subsystem	Communications
Class	ClientNetworkInterface
Service	Storage
Operation	Push
Prototype	*(void) ClientNetworkInterface::Push (string objectType, map<string, string>)*
Description	This operation takes a map of the properties of the object being modified and a string which will later be used to indicate which database table to modify. The data is passed from client to server, and then used for the Request Management service of the Server Controller.
Traceability	All use cases Appendix I (EXCLUDING UC-700 and EXTENDS use cases)

Subsystem	Communications
Class	ClientNetworkInterface
Service	Storage
Operation	Set Audit Time
Prototype	*(void) ClientNetworkInterface::Set Audit Time (Time time)*
Description	This operation takes a time object, which is sent from client to server. The Request Management service of the Server Controller is then used to store the audit time.
Traceability	UC-650

Server Controller

Request Management

This service take requests from the Communications subsystem and uses the Storage subsystem's services to fulfill those requests.

Subsystem	Server Controller
Class	ServerController
Service	Request Management
Operation	Pull
Prototype	*(List<map<string, string>>) ServerController::Pull (string objectType, map<string, string>)*
Description	This operation takes a map of properties to filter by and a string which will be used to indicate which database table to look in. The string and map are used to create a StorageObject (which is a simple wrapper of those fields). Then that data is sent to the Retrieval service of the Storage subsystem.
Traceability	All use cases Appendix I (EXCLUDING UC-700 and EXTENDS use cases)

Subsystem	Server Controller
Class	ServerController
Service	Request Management
Operation	Create
Prototype	*(int unique id) ServerController::Create (string objectType, map<string, string>)*
Description	This operation takes a map of the properties of the object being created and a string which will be used to indicate which database table to add to. The string and map are used to create a StorageObject (which is a simple wrapper of those fields). Then that data is sent to the Storage service of the Storage subsystem.
Traceability	All use cases Appendix I (EXCLUDING UC-700 and EXTENDS use cases)

Subsystem	Server Controller
Class	ServerController
Service	Request Management
Operation	Push
Prototype	*(void) ServerController::Push (string objectType, map<string, string>)*
Description	This operation takes a map of the properties of the object being modified and a string which will be used to indicate which database table to modify. The string and map are used to create a StorageObject (which is a simple wrapper of those fields). Then that data is sent to the Storage service of the Storage subsystem.
Traceability	All use cases Appendix I (EXCLUDING UC-700 and EXTENDS use cases)

Subsystem	Server Controller
Class	ServerController
Service	Request Management
Operation	Set Audit Time

Prototype	*(void) ServerController::Set Audit Time (Time time)*
Description	This operation takes a time, which is used to modify the Audit timer so that it triggers at the new time.
Traceability	UC-650

Subsystem	Server Controller
Class	ServerController
Service	Request Management
Operation	Get Audit Time
Prototype	*(Time time) ServerController::Get Audit Time ()*
Description	This operation returns the current time that the audit is set to take place at.
Traceability	UC-650

Storage

Storage

This service allows new data to be added to the database, or existing data to be modified.

Subsystem	Storage
Class	Repository
Service	Storage
Operation	Create
Prototype	*(int unique id) Repository::Create (StorageObject storageObject)*
Description	This operation take a storage object and constructs an SQL query, which it then runs on the SQLite database. In this case, the query is an insert statement. The storage object contains a string which indicates the name of the table, and a map from column names to values. After the insert statement, this subsystem gets the unique id which was selected automatically by the database for the new object, and returns it.
Traceability	All use cases Appendix I (EXCLUDING UC-700 and EXTENDS use cases)

Subsystem	Storage
Class	Repository
Service	Storage
Operation	Push
Prototype	*(void) Repository::Push (StorageObject storageObject)*
Description	This operation take a storage object and constructs an SQL query, which it then runs on the SQLite database. In this case, the query is an update statement. The storage object contains a string which indicates the name of the table, and a map from column names to values.
Traceability	All use cases Appendix I (EXCLUDING UC-700 and EXTENDS use cases)

Retrieval

This service allows data from the database to be retrieved and returned.

Subsystem	Storage

Class	Repository
Service	Retrieval
Operation	Pull
Prototype	*(List<StorageObject>) Repository::Pull (StorageObject storageObject)*
Description	This operation take a storage object and constructs an SQL query, which it then runs on the SQLite database. In this case, the query will be a select statement. The storage object contains a string which indicates the name of the table, and a map from column names to values (to filter by). Each row returned is read into a map of column names to values which is used to create a storage object. These objects are placed in a list and returned.
Traceability	All use cases Appendix I (EXCLUDING UC-700 and EXTENDS use cases)

Audit Process

This service uses an SQL query to update 'pending' status follow-ups to 'overdue' status where appropriate.

Subsystem	Storage
Class	Repository
Service	Audit Process
Operation	Run Audit
Prototype	*(void) Repository::Run Audit (Date today)*
Description	This operation runs an SQL query which finds all followups with a due date earlier than today and a status of pending, and updates their status to overdue.
Traceability	UC-700

5 Class Interfaces

The class interfaces section provides a description of the classes involved in providing operations for each service. For each class a UML class diagram is provided, specifying its attributes and the operations involved in the service.

ReportsManagement (Reports Management Subsystem)

The ReportsManagement class is the sole class in its subsystem. The class provides the interface for the operations that makes up the Report Management service, used by the GUI subsystem. For additional detail on this class and its subsystem, please refer to the System Decomposition section (2.2) of this document.

ReportsManagement
-reportType : ReportTypeEnum -filters : vector<Filter> -reportParameters : map<string, string>
+setReportType(ReportType reportType) : void +createFilter(Filter filter) : void +modifyFilter(Filter filter) : void +removeFilter(Filter filter) : void +setReportParameters(Parameters parameters) : void +generateReport() : Report +saveReport(filestream file) : void

AccessControl (Access Control Subsystem)

The AccessControl class is the sole class in its subsystem. The class provides the interface for the operations that makes up the Login Verification service, used by the GUI subsystem. For additional detail on this class and its subsystem, please refer to the System Decomposition section (2.2) of this document.

AccessControl
+verifyUsername(string username) : LoginStatus +getLoginStatus() : LoginStatus

PatientData (Patient Data Management Subsystem)

The PatientData class is the sole class in its subsystem. The class provides the interface for the operations that makes up the Patient Management and the Physician Retrieval services, both of which are used by the GUI subsystem. For additional detail on this class and its subsystem, please refer to the System Decomposition section (2.2) of this document.

PatientData
+getPatientList(FilterType filterType, Physician physician, FollowupType followupType) : List<Patient> +getFullPatient(Patient patient) : Patient +createPatient(Patient patient) : void +modifyPatient(Patient patient) : void +createConsultation(Consultation consult) : void +modifyConsultation(Consultation consult) : void +createFollowup(Followup followup) : void +modifyFollowup(Followup followup) : void +getPhysicianList() : List<Physician>

UserData (User Data Management Subsystem)

The UserData class is the sole class in its subsystem. The class provides the interface for the operations that makes up the User Management service, used by the GUI subsystem. For additional detail on this class and its subsystem, please refer to the System Decomposition section (2.2) of this document.

UserData
+createPhysician(Physician physician) : void +modifyPhysician(Physician physician) : void +createAdminAssistant(AdminAssistant adminAssistant) : void +modifyAdminAssistant(AdminAssistant adminAssistant) : void +createSysAdmin(SysAdmin sysAdmin) : void +modifySysAdmin(SysAdmin sysAdmin) : void

SystemAdministration (System Administration Subsystem)

The SystemAdministration class is the sole class in its subsystem. The class provides the interface for the operations that makes up the Network Settings Management service, used by the GUI subsystem. For additional detail on this class and its subsystem, please refer to the System Decomposition section (2.2) of this document.

SystemAdministration
-loginStatus : LoginStatusEnum
+setClientIPAddress(string ipAddress) : void +setClientPortNumber(int portNumber) : void +setAuditTime(Time time) : void +getAuditTime() : Time

ClientData (Client Data Storage Subsystem)

ClientData is a singleton facade which provides access to the services offered by the Client Data Storage subsystem. It keeps a Factory object, a Warehouse object, and a NetworkSettings object. The create, push, and setAuditTime operations of the Storage service are passed to the Factory class. Similarly, data retrieval operations of the Retrieval service, such as pull requests to the database for specified model objects, are passed to the Warehouse object. Finally, the setIPAddress and setPortNumber operations are passed off to the NetworkSettings class, representing the Network Settings Storage service.

ClientData
-Factory factory -Warehouse warehouse -NetworkSettings networkSettings
+T getModelObject<T>(int id) +list<int> pull(string className, map<string, string> filter) +list<int> getReportResults(ReportType type) +Time getAuditTime() +void create(ModelObject *object) +void modify(ModelObject *object) +void setAuditTime(Time time) +void setIPAddress(string ip) +void setPortNumber(int port)

Object Interpreter (Object Interpreter Subsystem)

ObjectInterpreter does not keep any state. It simply provides functions for two services. The getPropertyMap function fulfills the Object to Property Map service, while the getModelObject function fulfills the Property Map to Object service.

The purpose of the ObjectInterpreter is to abstract away the functionality of translation from the Client Data Storage and Communications subsystems. Neither of those subsystems need to know how translation works, and only one of them need employ it. The design calls for the Client Data Storage subsystem to make use of the services provided by the ObjectInterpreter class. By abstracting these services into their own class and subsystem, we have the freedom to change the method of translation without changing Client Data Storage or Communications.

Object Interpreter
+Map<string, string> getPropertyMap (ModelObject *modelObject) +void getModelObject (map<string, string>, ModelObject *returnObject)

ClientNetworkInterface (Communications Subsystem)

ClientNetworkInterface is an interface into the Communications subsystem. Its operations provide the Retrieval and Storage services. ClientNetworkInterface contains the IP address and port of the cuCare server, and is used to create a connection and make requests, using TCP sockets. Each of its operations (the ones relevant to its services), upon being called, makes the appropriate request to the server, and, if necessary, populates an output pointer that has been passed to it. Each operation also provides error detection, and in the event that a problem occurs during the request, it will populate a passed string with an appropriate error message.

ClientNetworkInterface
-serverIP : QHostAddress -serverPort : quint16
+create(in objectType : string, in pObjectMap : Map<string, string>*, out pReturnID : int*, out pErrorString : string*) : bool +push(in objectType : string, in pObjectMap : Map<string, string>*, out pErrorString : string*) : bool +pull(in objectType : string, in pFilterMap : Map<string, string>*, out pMapList : List<Map<string, string>*, out pErrorString : string*) : bool +setAuditTime(in auditTime : Time, out pErrorString : string*) : bool +retreiveAuditTime(out auditTime : Time*, out pErrorString : string*) : bool

ServerController (Server Controller Subsystem)

Through its operations, the ServerController class provides the Request Management service to ServerNetworkRequestInterpreter (a class within the Communications subsystem). The operations involved are used to retrieve information from, and store information in persistent storage (by making use of the Storage subsystem). In addition to providing an interface into storage, ServerController also provides operations for getting and setting the daily audit time (also part of the Request Management service).

ServerControler accesses storage using a Repository object that it keeps as a data member. It also keeps the AuditTimer object that triggers the running of audits. When an audit is triggered, ServerController makes the necessary updates to storage using its Repository. Dealing with storage access using this Repository is the main role that ServerController plays, and if, in the future, more functionality is added to the server component of cuCare, ServerController would likely be the class responsible for it.

It is worth noting that the ServerController class implements the “singleton” design pattern. Additional information on this can be found in the Design Patterns section of this document under the Singleton sub-heading.

ServerController
-storageRepo : Repository -auditor : AuditTimer
+create(in objectType : string, in pObjectMap : Map<string, string>*, out pReturnID : int*, out pErrorString : string*) : bool +push(in objectType : string, in pObjectMap : Map<string, string>*, out pErrorString : string*) : bool +pull(in objectType : string, in pFilterMap : Map<string, string>*, out pMapList : List<Map<string, string>*, out pErrorString : string*) : bool +setAuditTime(in auditTime : Time, out pErrorString : string*) : bool +getAuditTime(out auditTime : Time*, out pErrorString : string*) : bool

Repository (Storage Subsystem)

The Repository class of the Storage subsystem provides three services to the ServerController. The create and push functions make up the Storage service. The pull function provides the retrieval service. The runAudit function provides the Run Audit service. The repository class keeps a pointer to a *Database*. This pointer is currently always instantiated with an SQLite3Database object. Repository translates requests to its services into SQL queries, which it sends to the Database object.

Repository
-Database* db
+bool pull (string table, StorageObject filters, list<StorageObject>* returnObjects) +bool push (string table, StorageObject values) +bool create (string table, StorageObject values, int &uid) +bool runAudit ()

Appendix I – Traceability Reference

This section has been included to facilitate traceability of the system design elements to use cases developed for the requirements analysis assignment. The list of all of cuCare use cases, including a short description for each one, is presented below.

Identifier	Name	Description
	SYSTEM ACCESS	
UC-100	**LogIn**	User (any type) logs on to the system
UC-110	**LogOut**	User (any type) logs out of the system
	PATIENT MANAGEMENT	
UC-200	**CreatePatient**	AdminAssistant or Physician creates a new patient profile
UC-210	**ViewPatient**	AdminAssistant or Physician views an existing patient record
UC-220	**ModifyPatient**	AdminAssistant or Physician makes changes to an existing patient record
UC-230	**RemovePatient**	AdminAssistant or Physician removes a patient record from the system
	CONSULTATION MANAGEMENT	
UC-300	**CreateConsultation**	AdminAssistant or Physician schedules a new consultation
UC-310	**ViewConsultation**	AdminAssistant or Physician views an existing consultation record
UC-320	**ModifyConsultation**	AdminAssistant or Physician makes changes to an existing consultation record
UC-330	**PhysicianModifyConsultation**	Physician makes changes to an existing consultation record (describes additional functionality available only to Physicians)
UC-340	**CancelConsultation**	AdminAssistant or Physician cancels a consultation
	FOLLOW-UP MANAGEMENT	
UC-400	**CreateFollowup**	Physician creates a follow-up request
UC-410	**ViewFollowup**	AdminAssistant or Physician views a follow-up record
UC-420	**ModifyFollowup**	A general description of how a user would make changes to an existing follow-up record
UC-430	**EnterFollowupResults**	AdminAssistant or Physician enters the results of a medical test or a specialist diagnosis into a follow-up record
UC-440	**PhysicianModifyFollowup**	Physician makes changes to a follow-up record
UC-450	**MarkFollowupComplete**	Physician marks a follow-up record as complete
UC-460	**CancelFollowup**	Physician cancels a follow-up request

	REPORT GENERATION	
UC-500	**RunReport**	AdminAssistant or Physician runs a statistical report
UC-510	**RunFollowupRankingReport**	AdminAssistant or Physician runs a report that provides a ranking of types of follow-up requests that are most often overdue
UC-520	**RunInactivePatientReport**	AdminAssistant or Physician runs a report that lists patients who have had no consultations for a specified amount of time
UC-530	**RunPatientConsultationCountReport**	AdminAssistant or Physician runs a report that lists patients with a high number of consultations
UC-540	**RunOverdueFollowupCountReport**	AdminAssistant or Physician runs a report that lists patients with a high number of overdue follow-ups
UC-550	**RunBillingCounsultationsCountReport**	AdminAssistant or Physician runs a report that lists the total number of consultations associated with each billing number
UC-560	**RunFollowupStatusReport**	Physician runs a report that lists patients filtered by follow-up status
UC-570	**AddReportFilter**	AdminAssistant or Physician applies a filter to a report they are currently viewing
UC-580	**ModifyReportFilter**	AdminAssistant or Physician makes changes to a filter that is being or will be applied to a report they are currently viewing
UC-590	**GenerateReport**	Having selected the relevant report options and applied any of the desired filters, AdminAssistant or Physician generates the desired statistical report
UC-595	**SaveReport**	AdminAssistant or Physician saves a reports they have generated to a text file that can be accessed outside of the cuCare system
	SYSTEM ADMINISTRATION	
UC-600	**CreateAccount**	SysAdmin creates a new user account
UC-610	**ViewAccount**	SysAdmin views an existing user account
UC-620	**ModifyAccount**	SysAdmin makes changes to an existing user account
UC-630	**RemoveAccount**	SysAdmin removes a user account from the system
UC-640	**ViewSystemConfiguration**	SysAdmin views configuration settings for the cuCare client application
UC-650	**SetAuditTime**	SysAdmin changes the time of the daily audit process
UC-660	**SetServerIP**	SysAdmin changes the IP address of the cuCare server
UC-670	**SetServerPort**	SysAdmin changes the port number of the cuCare server
UC-680	**TestConnection**	SysAdmin tests the connection to the cuCare server using current system settings

	AUTOMATION	
UC-700	**RunAudit**	Execution of the daily audit process
	EXTENDS	
UC-800	**SelectPhysician**	Extends use cases involving data entry by providing a list of available physicians
UC-810	**ServerResponseTimeout**	Extends use cases involving client server communication by providing the user with a prompt to inform then that the cuCare client application has failed to establish a connection with the server
UC-820	**InvalidUsername**	Extends the **LogIn** use case in instances where an invalid username has been entered in an attempt to log in to the system
UC-830	**InvalidDate**	Extends use cases involving data entry by notifying the user that the date they have entered is invalid
UC-840	**InvalidTime**	Extends use cases involving data entry by notifying the user that the time they have entered is invalid
UC-850	**UsernameNotAvailable**	Extends the CreateAccount use case in instances where a username is already being used