Carleton University

CUBook
System and Object Design Document (Revised)

Team Phoenix
Andrew Thompson
Sina Keang
Farshad Muhammad
Aaron Katz

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

Table of Content

1 Introduction

The main use of the System and Object Design document is more of a phase for the development team. Here we look for system design models from the CuBook non functional requirements, analysis object model and dynamic model. The non functional requirements we are interested in is the constraint of the CuBook system requirement for handling at least 6 clients connections at once. The constraint of the flashfeed showing a new newsflash in constant time. For the CuBook system, we consider constant time to be 2 secs after posting a newsflash.

For CuBook we have decided to use two architectural style. The overall system will use a Client/Server architect to allow multiple client connections. Inside the client layer, a Model-View-Control style will be used to allow navigation between multiple windows consisting of the user profile window, the flashfeed window, the segment window and the filter window.

1.1 Purpose of System

The purpose of this document is to properly detail all the subsystems, show their decompositions, their relations and what they consist and contain. Our system CuBook is split into two sides. A client side and a server side. The client side is where most of the data handling and user interactions happen. It is also where the user connects to the server side from. The client subsystems provide such services such as taking input, handling input, creating objects from input and checking data. The client side also keeps a back up called the client cache from which the graphical user interface can take information from without requesting it from the server.

The server side of the system provides storage of persistent data. It also controls connections and manages traffic as well as ensuring all users are notified about persistent object changes, removal and creation. The main storage is handled and done by SQL based page tables which are controlled by wrapper classes.

Both the server and client sides of the system communicate with each other through TCP/IP socket connections which send a byte array containing a specified message protocol (explained further on). Most of the subsystems in this system are independent and promote loose coupling and high cohesion.

1.2 Design Goals

Performance Criteria	Descriptions	Traceability
Throughput:	CuBook system must be able to handle 6 or more clients connections to it at once without taxing performance time.	NR 7.30
Response Time:	Common tasks(Sending a post, reading a post..etc) should not exceed one minute.	NR 7.120

Dependability Criteria	Descriptions	Traceability
Robustness	System should not crash due to invalid user input. Invalid user input must be properly handled allowing user to continue using the system. Data must be properly persisted to prevent corruption from occurring.	NR 7.170, NR 7.171, NR 7.160
Availability	CuBook system should be responsive and available to the user from the time user logs in to the time the user logs out 90% of the time.	NR 7.171, NR 6.90
Fault Tolerance	During data corruption, CuBook should avoid corrupting or crashing other processes or forms of data. Data corruption should be handled by the system and the user should be notified about the corruption of the data. User must be notified of all major functional errors. Data should be backed up to allowed recovery in the event of data loss. If network unexpectedly disconnect, the client should attempt to reconnect up to three times before notifying the user of lost connection.	NR 7.170, NR 7.161

Maintainability Criteria	Descriptions	Traceability
Security	All layers containing subsystems in the system should be properly separated and should be as loosely coupled as possible. All users must be running a separate client process and all client processes should be separate so as to not have any data leakage or overlap. Users must have a unique authenticated login for the system to be able to identify them and grant them individual access to their information. Client and server processes must be run on separate machines. Users can only make newsflashposts in accordance to newsflashboards they are subscribed to. Only users with instructor type accounts can cause removal of persistent data i.e newsflashposts and segments, from the system. Users of the system should not have access to other users' information unless they're allowed by the system (e.g. Instructors have access to view, but not update, the information of the the students they teach).	NR 7.71, NR 7.60, NR 7.130

Cost Criteria	Descriptions	Traceability
Development Cost	The time required to document, analysis and develop the CuBook system should not exceed a period of three months.	NR. 695

End User Criteria	Descriptions	Traceability
Usability	User interface should be easy to navigate. User interface should look professional. Users should be able to easily identify other users by their avatars. Users must be able to save their filters to be used in other sessions. System should not have to be recompiled every time the IP and port of the server is changed. Users should not have to reenter valid information upon a validation or formatting error.	NR 7.20, NR 7.40, NR 7.50, NR 7.150, NR 7.30, NR 7.60, NR 7.101, NR 7.110

2. System Design

2.1 Overview

In the following section you will find a full decomposition of our system. This includes in order; a breakdown of our system into subsystem, diagrams outlining their relationship. Our design strategies, with diagrams and dialog to explain themselves a further breakdown of our subsystems into services and finally the protocol to which messages will be sent throughout the system, both internally and across the server – client.

2.2 Subsystem decomposition

Subsystems	Descriptions	Traceability
CuBook Client Subsystem	The client layer of the client server architectural style	NR 7.60, NR 7.130
CuBook Server Subsystem	The server layer of the client server architectural style	NR 7.70
UI Handler Control Subsystem	Handles all input from user and all output from CuBook system. Gives a graphical interface for all messages and interactions with the user (as well as handled exceptions).	NR 2.10, NR 7.20, NR 7.150, NR 7.170, NR 7.30, NR 1.10, NR 1.20
UI View Subsystem	The forms of the User interface	NR 7.20
Client Cache Model Subsystem	Maintains an active memory of important information that can be readily accessed by the client. In case of communication disconnection the end user will not lose their data and can still have limited access to their current page.	NR 1.170, NR 7.30, NR 6.00, NR 1.10, NR 1.20, NR 7.10
Client Communication Manager Subsystem	Queues all data trying to go to or come from the server and sends it to the server or its appropriate handling subsystem. Acts as a gateway converting data to and from the byte arrays for transfer over to the server.	NR 7.90, NR 7.30, NR 7.60, NR 7.80, NR 7.130, NR 7.101
Server Communication Manager Subsystem	Queues all data trying to go to or come from each client and sends it to the intended client(s) or its appropriate handling subsystem. Acts as a gateway converting data to and from the bit array for transfer over to the Client	NR 7.90, NR 7.120, NR 7.30, NR 7.60, NR 7.80, NR 7.130, NR 7.110
Database Management Subsystem	Holds all persisted data as well as ID's connected to the data. Database have page tables where all the data is stored and is mainly based on SQL. Manages all information going to and coming from the database, ensuring that it is mapped to the right area, and accessing what is being asked for by the handler subsystems	NR 7.101, NR 7.110, NR 7.70, NR 7.160, NR 7.161, NR 7.10

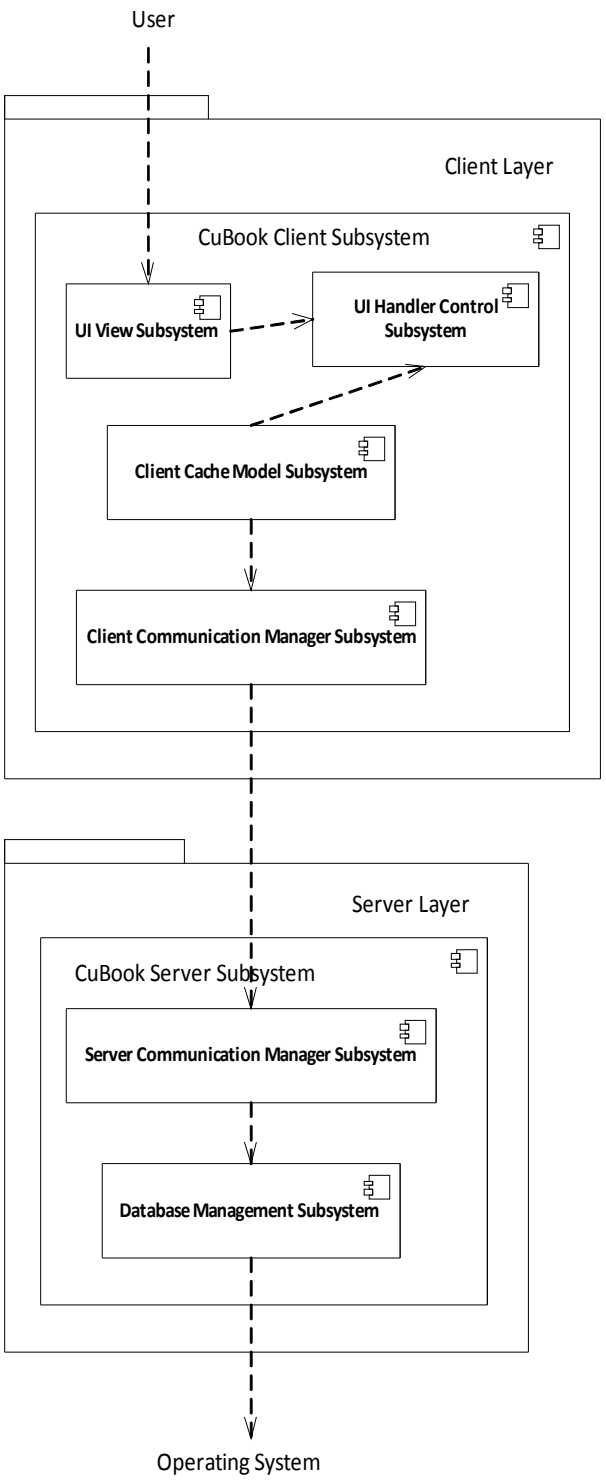


Figure 1: High Level Component Diagram
Of CuBook Client/Server Architectural Style
Dependency arrows are shown to indicate
Loose Coupling.

CuBook use a Client/Server style with
The CuBook Client subsystem being the client layer
And the CuBook Server subsystem being the Server layer

The Clients know the interface of the Server
The server does not need to know the interface of the client.
End users interact only with the client. This is the reason we are
Using Client/Server architect as our overall high level style

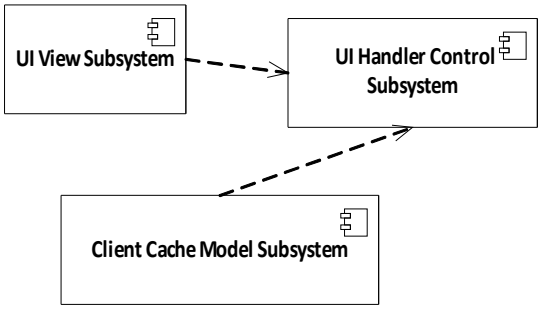


Figure 2: The Client Layer contains a MVC style architectural Connecting to the Client Communication Manager subsystem. The UI View Subsystem represent the “View” while the UI Handler Control subsystem represent the “Control” and finally the Client Cache Model Subsystem represent the “Model” of the Model – View – Control Style.

Using the MVC decouples the entity objects.

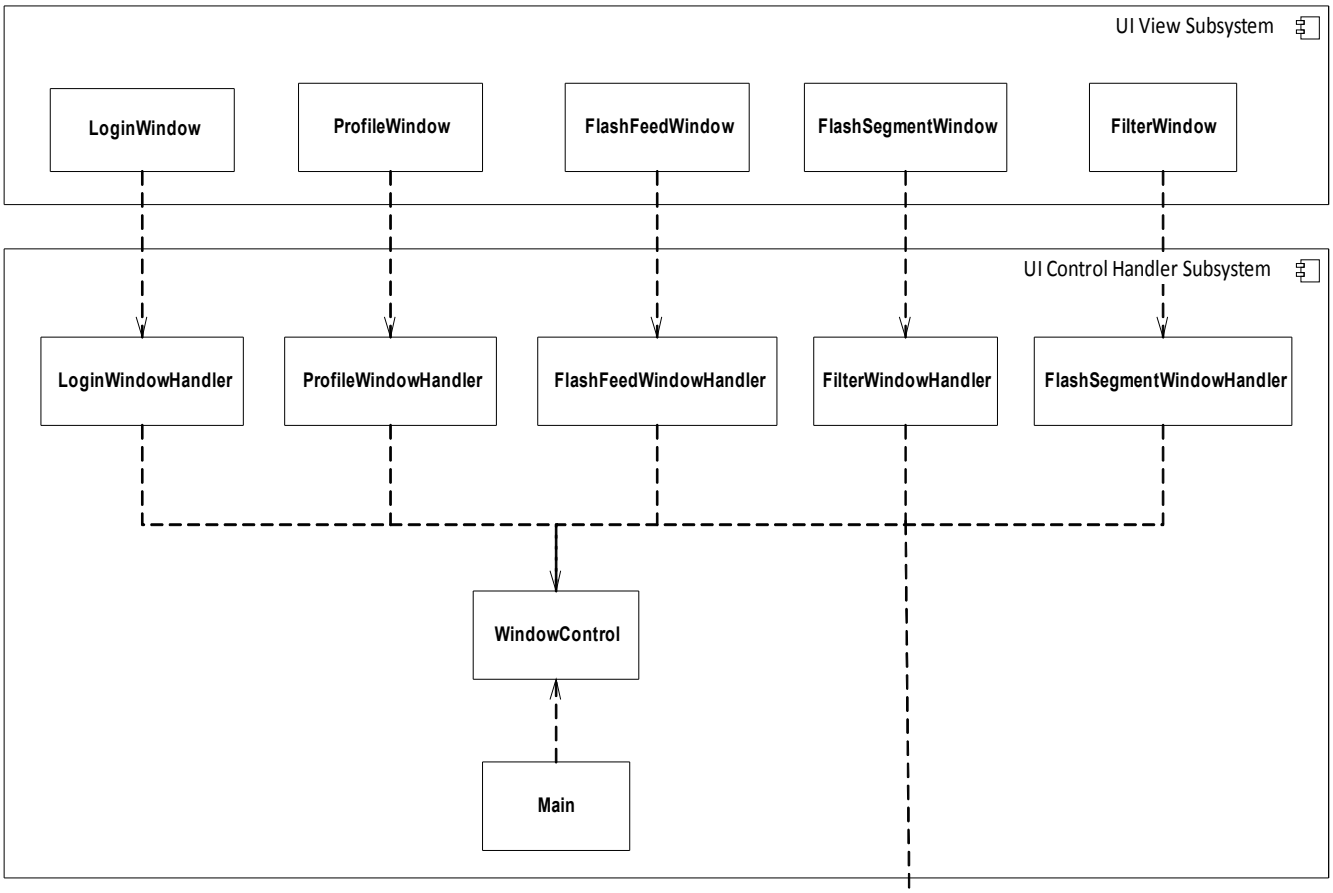
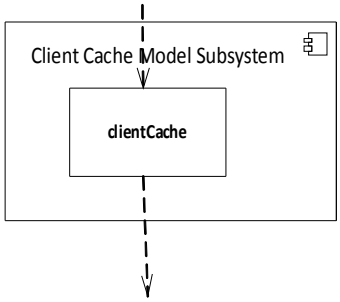


Figure 3A: Detailed Component Diagram of the CuBook Client Subsystem
Showing all classes in the UI View, UI Control Handler, and
The Client Cache Model Subsystems.
Dependency arrows are shown to indicate Loose Coupling.



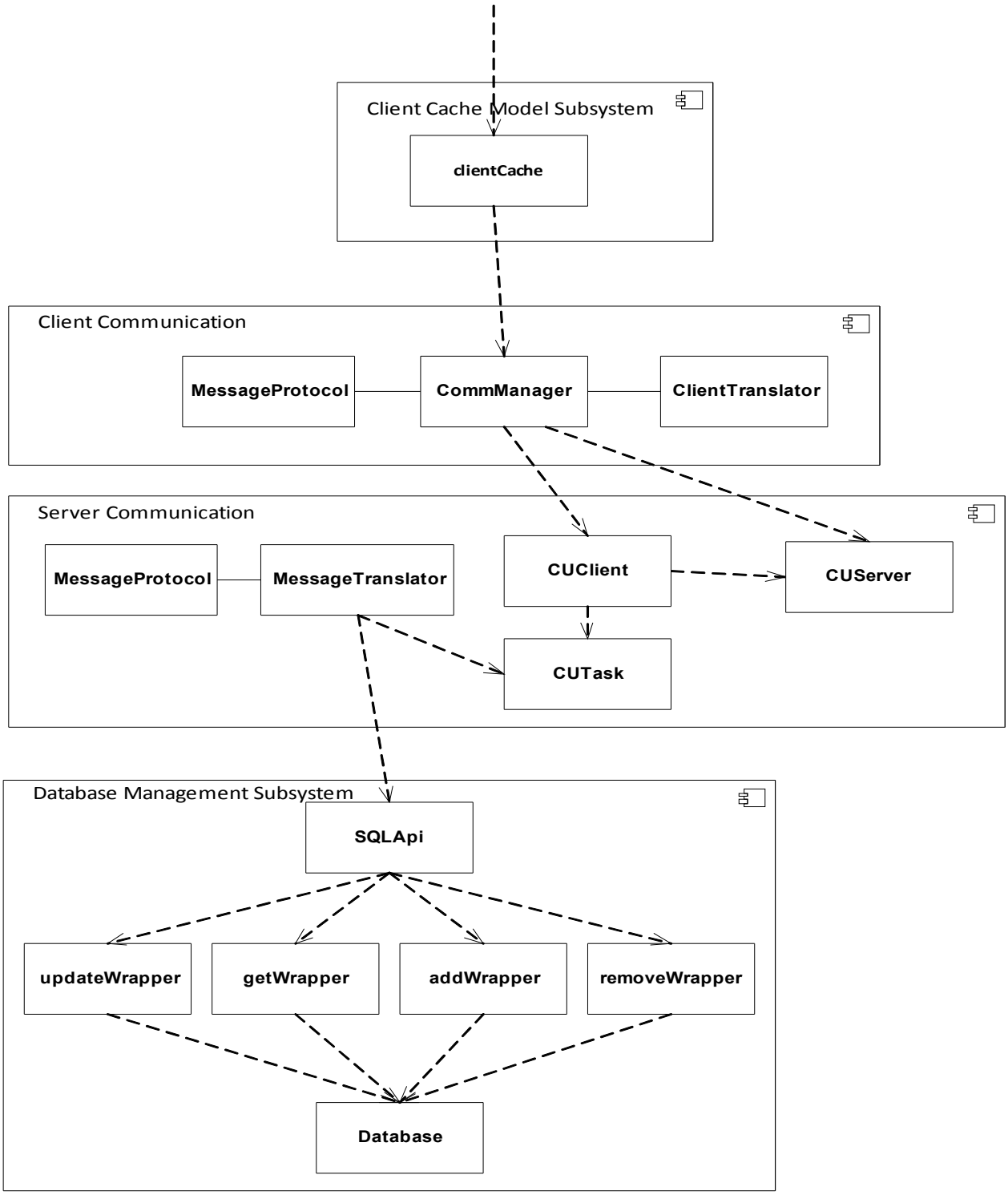


Figure 3B: Detailed Component Diagram of the CuBook Server Subsystem showing all classes in the Client Communication, Server Communication, and the Database Management Subsystems.
Dependency arrows are shown to indicate Loose Coupling.

2.3 Design Strategies

2.3.1 Hardware/Software Mapping

There are two main nodes in CuBook, the Client Machine node and the Server Machine node. The two nodes talk to each other through the use of a TCP/IP connection. The Client Communication Manager Component in the Client Machine node connect to the Server Communication Manager Subsystem in the Server Machine node using this TCP/IP connection.

Nodes	Components	Subsystems of Component	Traceability
ClientMachine	CuBook Client Subsystem	UI Handler Control Subsystem, Client Cache Model Subsystem, UI View Subsystem, Client Communication Manager Subsystem	NR7.60, NR7.80
LambdaServer	CuBook Server Subsystem	Server Communication Manager, Database Management Subsystem	NR 7.70, NR7.80
ClientMachine	UI View Subsystem	Only contain classes	NR 7.20
ClientMachine	UI Handler Control Subsystem	Only contain classes	NR 2.10, NR 7.20, NR 7.150, NR 7.170, NR 7.30, NR 1.10, NR 1.20
ClientMachine	Client Cache Model Subsystem	Only contain classes	NR 1.170, NR 7.30, NR 6.00, NR 1.10, NR 1.20, NR 7.10
ClientMachine	Client Communication Manager Subsystem	Only contain classes	NR 7.90, NR 7.30, NR 7.60, NR 7.80, NR 7.130, NR 7.101
LambdaServer	Server Communication Manager Subsystem	Only contain classes	NR 7.90, NR 7.120, NR 7.30, NR 7.60, NR 7.80, NR 7.130, NR 7.110
LambdaServer	Database Management Subsystem	Only contain classes	NR 7.101, NR 7.110, NR 7.70, NR 7.160, NR 7.161, NR 7.10

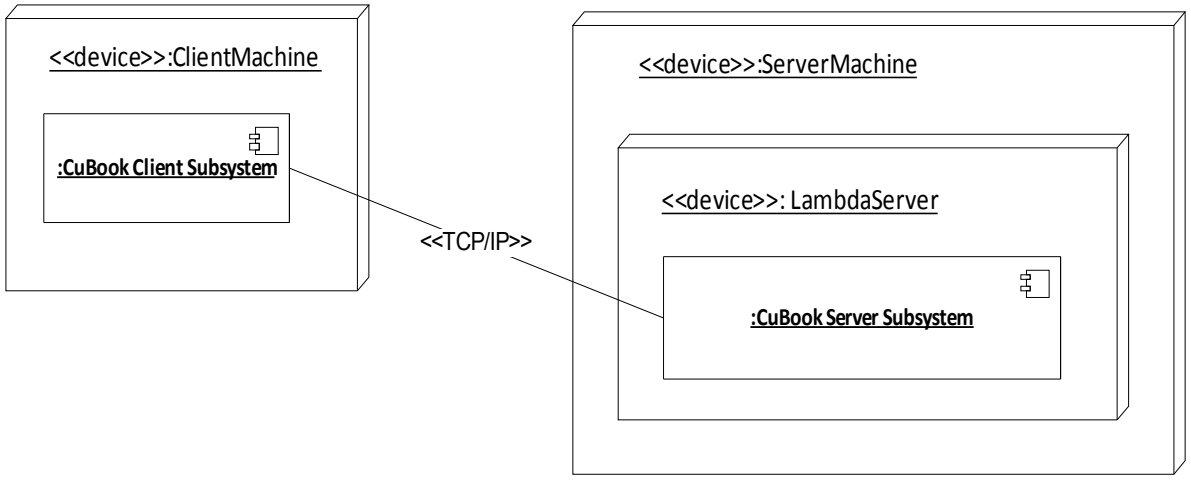


Figure 4: High Level Deployment Diagram
Showing connection between ClientMachine and ServerMachine nodes

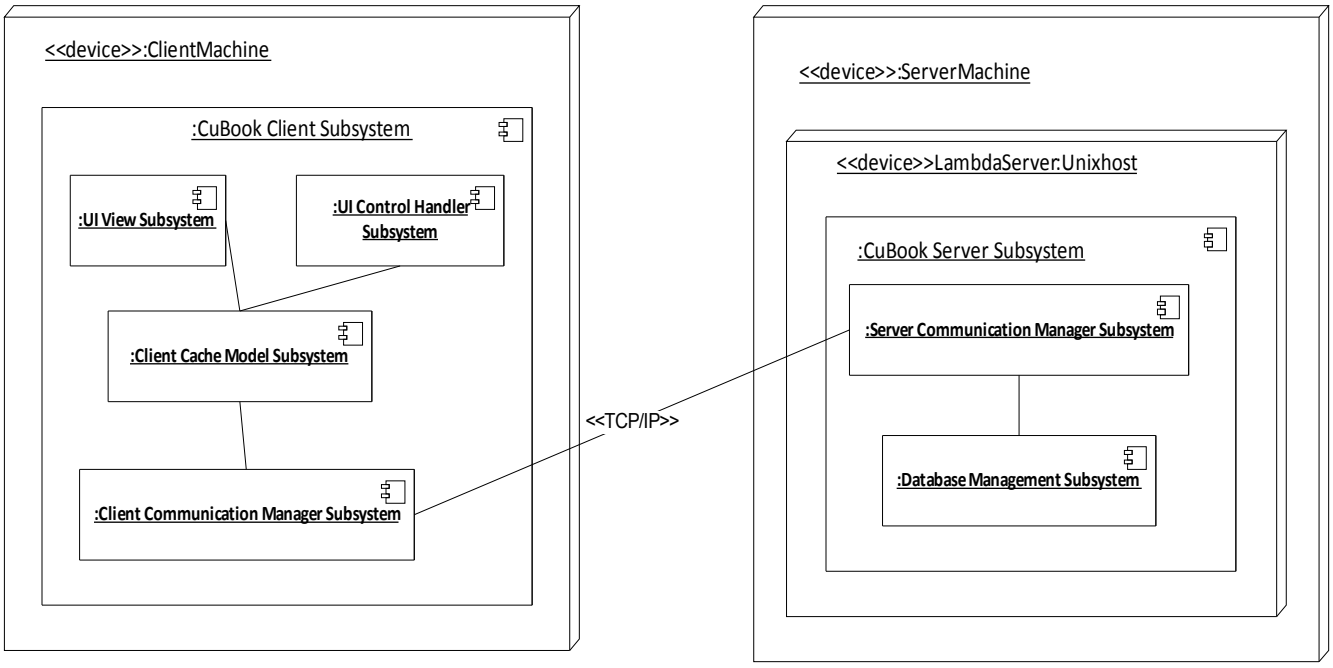


Figure 5: Detailed Deployment Diagram
Showing all subsystems in the ClientMachine and ServerMachine nodes

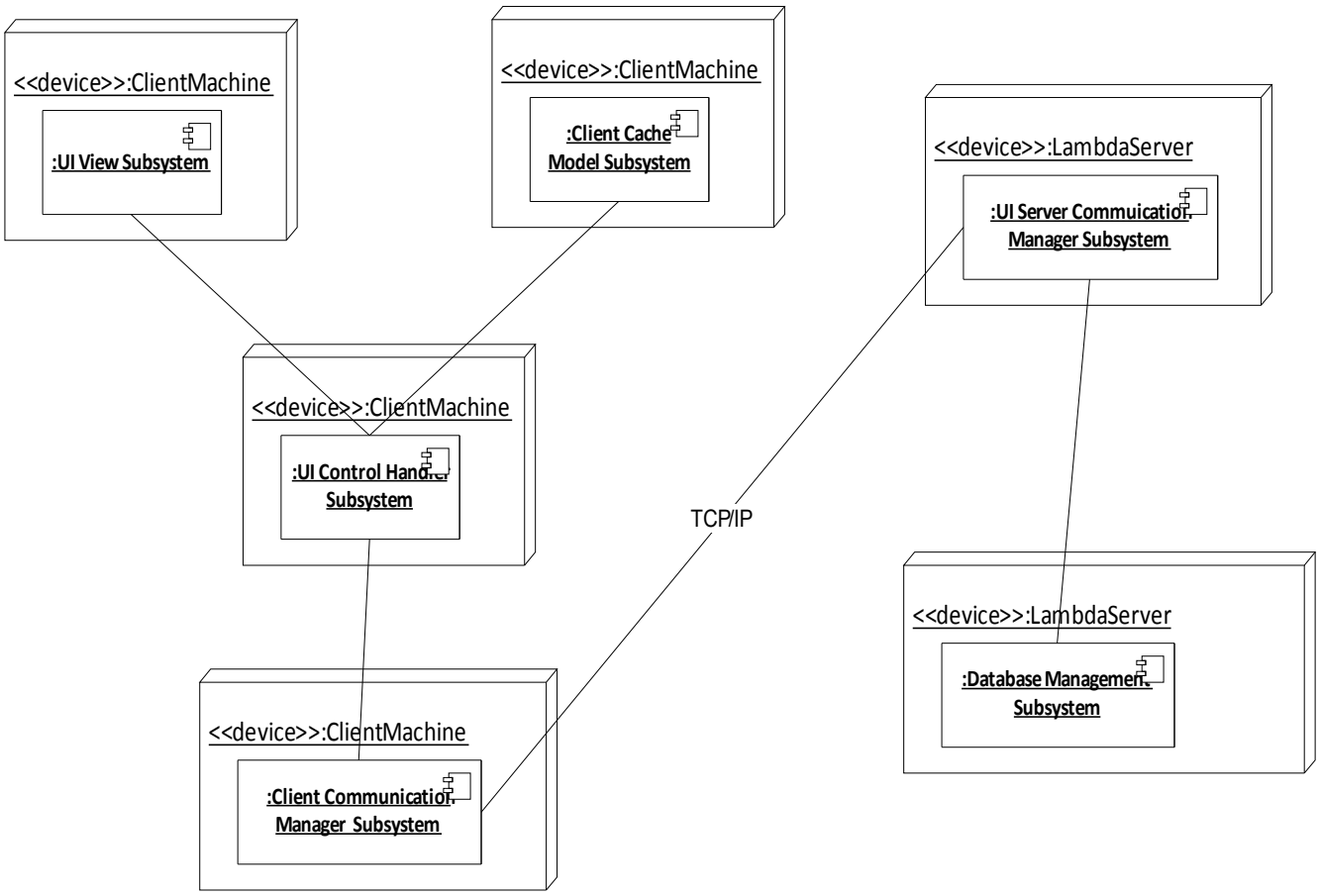


Figure 6: Deployment Diagram representing the allocation of Components to different nodes .

2.3.2 Persistent Data Management

Persistent data will be stored on the server machine by the use of SQLAPI in the Database management subsystem. This makes the database easily maintainable and is fault tolerant. Organization in the database is through model tables.

The objects in persistent storage:

newsFlash – posts made on the FlashFeed or FlashBoards or FlashSegment by the users or accessed by users.

FlashBoard – Predefined course entity objects already in the database for all courses offered in Carleton University

FlashSegment – flash segment created and viewed by users for courses.

UserProfile – profile information users enter into the system and also some predefined information for them, such as student Ids.

Filter- User filters for each session.

CUPicture – pictures uploaded as profile photo or uploaded by newsFlash as attachment

CUUrl- links saved by newsFlash as attachment

IDs – All forms of IDs are put into database. They can be used as keys linked to other entity objects and also have their own model page tables. The different type of ID's stored are

-user_id
-newsflash_id
-flashboard_id
-flashsegment_id
-attachment_id
-link_id

2.3.3 Access Control and Security

Our major access control and security strategy is to limit access on the client side that is to say the operations that the Instructor can access and Students cannot. Are not available to the students in the user interface (not click able). In addition, whenever a call is made for a security sensitive operation proper identification is required. I.E if a segment is deleted, the internal operation requires an Instructor ID which is specified by the system for Instructor type accounts only, and is not changeable. Both Student and Instructor accounts have different profile types. Each account type has its own profile information page and has items that the user cannot change, which are predetermined by the system. I.E Instructor and Student ID for login etc etc.

Object Actors	FlashBoard	newsFlash	FlashSegment	UserProfile	CUPicture	CUUrl
Student	getName getSegments isFiltered getFlashboardID setFlashBoardID setName addSegment setFiltered	getNewsFlashID getAuthorID getFlashBoardID getSegmentID getPictures getURLs getText getAuthorTag setNewsFlashID setAuthorID setFlashBoardID setSegmentID setAuthorTag addPicture addURL	getSegment getFlashboardID getName setSegment setFlashboardID setName	getUserID getUserType getNickname getRealName getLoginID getPicture setUserID setUserType setNickname setRealname setLoginID setPicture	getFormat getPicture setFormat setPicture toBytes	getLink getPageName setLink setPageName
Instructor	getName getSegments isFiltered getFlashboardID setFlashBoardID setName addSegment setFiltered	getNewsFlashID getAuthorID getFlashBoardID getSegmentID getPictures getURLs getText getAuthorTag setNewsFlashID setAuthorID setFlashBoardID setSegmentID setAuthorTag addPicture addURL	getSegment getFlashboardID getName setSegment setFlashboardID setName	getUserID getUserType getNickname getRealName getLoginID getPicture setUserID setUserType setNickname setRealname setLoginID setPicture	getFormat getPicture setFormat setPicture toBytes	getLink getPageName setLink setPageName

2.3.4 Global Software Control

Our CuBook system will be both event driven and multi-threaded. The server system will be multithreaded and will be able to handle up to 6 concurrent connections at once. This is done so that multiple client machines can operate through the same server machine and there is little delay or overlap between the processes of separate client machines. The server maintains a queue of all the client connection sockets and uses mutexes to handle concurrent client database call. Communications on a single socket will be event driven as the Event Queue will handle when a process has started and when it ends through signals encoded into the message protocols. This is done so that both client and the database in the server machines do not get flooded by requests from a single client and performance is maintained and memory is shared equally amongst all processes. This also minimizes data corruption and aids recovery through keeping logs of the Event Queue at any given time. Communication between subsystems and subsystem services on individual client and server machines will be event driven as subsystems will access each others services when needed (i.e. through user input to create objects).

2.3.5 Boundary Conditions

Subsystem Name	CuBook Client Subsystem
Entry Condition	CuBook Client Application is start up on the Client Machine
Flow of Events	UI Handler Control Subsystem starts up and populate the UI View Subsystem. UI Control Handler Subsystem then starts up the Client Communication Manager Subsystem and the ClientCache Manager Subsystem.
Exit Condition	CuBook Client Application is on the Client Machine

Subsystem Name	CuBook Server Subsystem
Entry Condition	CuBook Server Subsystem is start up on the Server Machine
Flow of Events	Server Communication Manager Subsystem and Database Management Subsystem is start up and waiting incoming connection from the client.
Exit Condition	CuBook Server Subsystem is shut down on the Server Machine

Subsystem Name	UI Control Handler Subsystem
Entry Condition	UI Control Handler Subsystem is called by the either the UI View Subsystem or the Client Cache Model Subsystem.
Flow of Events	UI View Subsystem call the UI Control Handler Subsystem which calls the the Client Cache Model Subsystem of what the UI View Subsystem is asking for.
Exit Condition	UI Control Handler Subsystem Calls Client Communication Manager Subsystem

Subsystem Name	UI View Subsystem
Entry Condition	User clicks on a boundary object on the GUI of CuBook
Flow of Events	That boundary object(such as send) calls the UI Handler Control Subsystem which is then sent to the Client Cache Model and Client Communication Manager Subsystem before coming back to the UI View Subsystem
Exit Condition	After a user input a command on the UI View Subsystem, the input is sent to the UI Handler Control Subsystem and

Subsystem Name	Client Cache Model Subsystem
Entry Condition	The CuBook Client Subsystem starts up
Flow of Events	Client Cache Model Subsystem starts up and Check with the Client Communication Manager Subsystem to access the CuBook for any update to the Database in the Database Management Subsystem.
Exit Condition	The CuBook Client Subsystem shut down

Subsystem Name	Client Communication Manager Subsystem
Entry Condition	Called by the Client Cache Model Subsystem
Flow of Events	Client Communication Manager Subsystem starts and tries to connect to the Server Communication Manager Subsystem for updates
Exit Condition	The CuBook Client Subsystem shut down

Subsystem Name	Server Communication Manager Subsystem
Entry Condition	CuBook Server Subsystem starts up
Flow of Events	Server Communication Manager Subsystem wait for incoming connection from the Client Communication Manager Subsystem. Data from the incoming connection is translated and send to the Database Management Subsystem and then send back to the Server Communication Manager Subsystem to be re translated and sent back to the Communication Manager Subsystem
Exit Condition	CuBook Server Subsystem is shut down on the Server Machine

Subsystem Name	Database Management Subsystem
Entry Condition	CuBook Server Subsystem starts up
Flow of Events	Database Management Subsystem checks for Data integrity and consistency. Then the subsystem awaits incoming call from the Server Communication Manager Subsystem
Exit Condition	CuBook Server Subsystem is shut down on the Server Machine
Configuration	Flashboard, Flashsegment, newsFlash, UserProfile, CUPicture and CUUrl are create/destroyed

Use Case Name	UCB 005.ClientSystemStartup
Entry Condition	CuBook Client system has been compiled and installed into client machine
Flow of Events	- CuBook Client system executable is started - CuBook Client system accesses IP and port of server through a flat file located in the installed system package. - Client establishes a connection with the server system. - Graphical interface of client is started. - CuBook Client system checks for any improper shutdown backups available.
Exit Condition	CuBook Client system shuts down

Use Case Name	UCB 004.ServerSystemStartup
Entry Condition	CuBook Server system has been compiled and installed into server machine
Flow of Events	- CuBook Server system executable is started - CuBook Server is started on the local machine through a specified port in a flat file located within the installed server package. - Server starts listening for client Connections.
Exit Condition	CuBook Server system service shuts down

Use Case Name	UCB 006.ServerSystemImproperShutdown
Entry Condition	CuBook Server System has been properly started and is listening for connections.
Flow of Events	- CuBook Server system faces an improper shutdown due to internal circumstances or accessing corrupt data. - Data on the server side that has not persisted must be sent to the database for immediate storage
Exit Condition	Crucial data has been persisted into database.

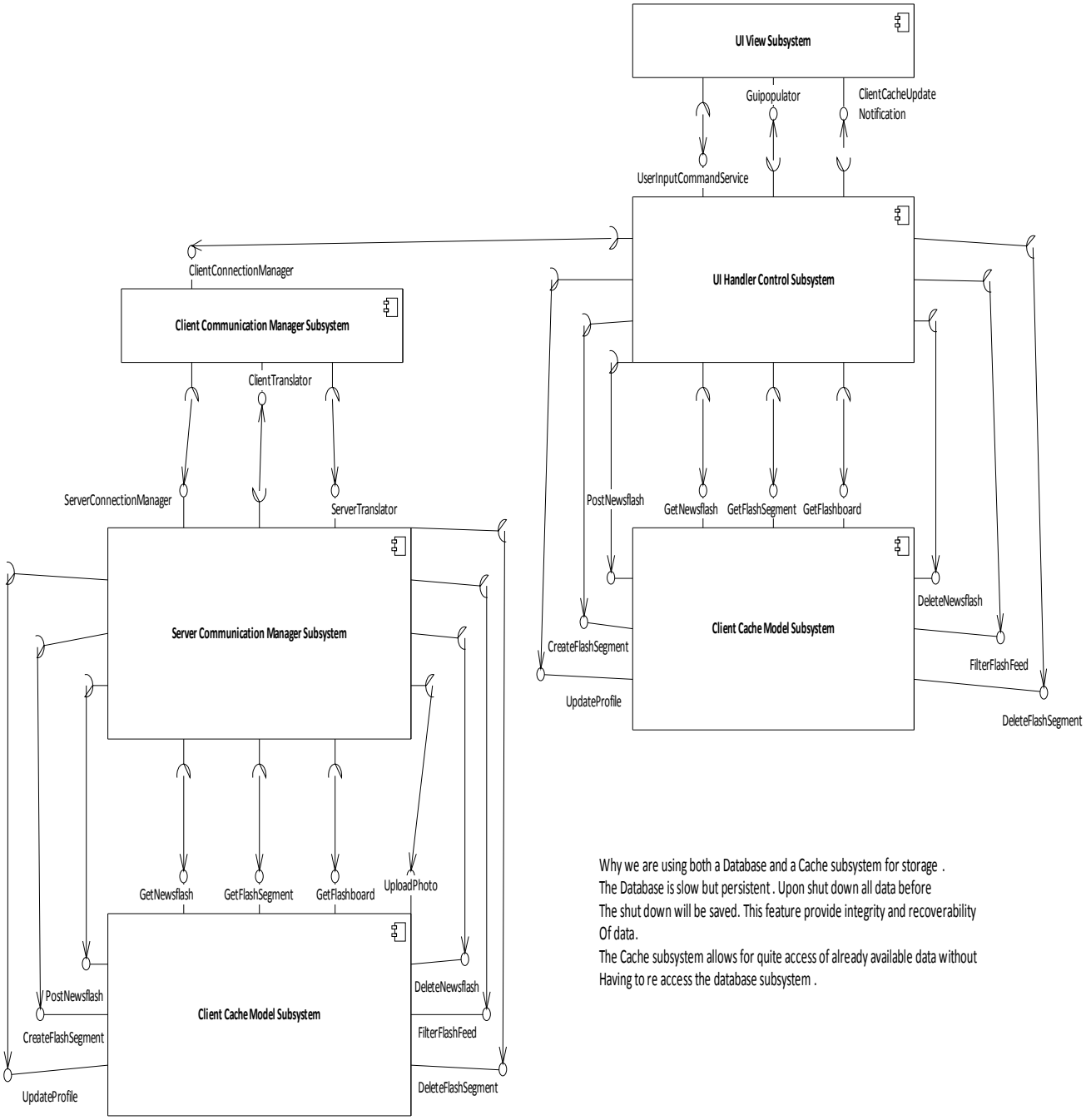
Use Case Name	UCB. 007.ServerSystemProperShutdown
Entry Condition	User has followed the steps and properly shutdown the server system
Flow of Events	- CuBook Server system notifies all client machines that it is about to shutdown. - CuBook Server system disconnects all client machines and closes all socket connections. - CuBook Server system finishes of all processes left in the server Event queue after all sockets have been disconnected -All data to be persisted has been sent to the database and has persisted. -Server machine shuts down and frees the port it was using.
Exit Condition	All persistent data has properly persisted in the database and all client machines have Disconnected.

Use Case Name	UCB. 008.ClientSystemImproperShutdown
Entry Condition	Client System is running and is connected to a server machine
Flow of Events	- Client machine is unexpectedly shutdown due to an error. - The data in the client cache is saved in a flat file to be accessed later for recovery. - A log of the event queue during imminent shutdown is kept in case processes need to be tracked.
Exit Condition	Client System has been exited after all data in the client cache have been saved and a log of the Event queue has been completed.

Use Case Name	UCB.009.ClientSystemProperShutdown
Entry Condition	User has followed all proper steps, and logged off their account.
Flow of Events	- All processes in the client event queue are sent over to the server side to be handled; persistent data is stored, or removed depending on the process. - Objects in the client cache are saved in case of emergency recovery. - CuBook Client system closes socket connection to the CuBook server system.
Exit Condition	The client executable is closed after log out.

Subsystem Use Case Name	UCBE 010.Unable to connect to server
Entry Condition	CuBook Client system has tried to connect to the server system but failed.
Flow of Events	- CuBook Client system tries 3 more times to connect to the server system using the specified IP and port. -CuBook Client system creates a log of its current event queue - CuBook Client system notifies the user that the IP and port specified are either unavailable or invalid and backs up its Client cache into a flat file.
Exit Condition	CuBook Client system has notified the use that the IP and port specified are either unavailable or invalid, or CuBook Client system establishes a connection within the three tries.

Subsystem Use Case Name	UCBE 011.ServerPortUnavailable
Entry Condition	The server system has specified a port and tried to start up its services.
Flow of Events	- Server tries to startup on a specified port from a flat file. - Server cannot startup its services due to the port being closed, in heavy use or it being blocked on the server machine.
Exit Condition	CuBook Server System notifies the user about the unavailability of the port.



Why we are using both a Database and a Cache subsystem for storage .
The Database is slow but persistent . Upon shut down all data before
The shut down will be saved. This feature provide integrity and recoverability
Of data.
The Cache subsystem allows for quite access of already available data without
Having to re access the database subsystem .

2.3.6 Design Patterns

Adapter – Subsystems using this design pattern:

Database Management Subsystem: This subsystem uses SQL based database implementation. Qt abstracts from original SQL design and thus uses legacy code to implement this form of database. SQL API class: This class sends translated data to be stored into the database. It also needs to have an understanding of how to handle SQL based objects. And thus uses legacy code in its implementation.

Client Communication Manager & Server Communication Manager Subsystems: These subsystems implement event Queues to handle connection based operations. They implement Queues from the STL vector class and thus use an abstraction of legacy code thus following the adapter style design pattern.

Observer- Subsystems using this design pattern:

Ui Handler Control Subsystem – This subsystem, populates a clients GUI according to their filter settings, filters hold subscription to certain types of data. For example if a course is in a user's filter, he will be subscribed to and thus be able to view changes and new NewsFlashPosts on their graphical interface.

MessageTranslator and CUServer– This subsystem notifies subscribed users about a new NewsFlashPost or segment that has been created so they can view it.

Abstract Factory – Subsystems using this design patterns:

The Client Communication Manager and Server Communicate Manager subsystems are an example of the usage of the abstract factory design pattern. They are completely mirrored subsystems (same classes and operations) but have a different flow of processes and communicate with different types of subsystems producing slightly varied results.

Fascade – Subsystems using this design pattern:

Ui Handler Control Subsystem – This large subsystem uses the fascade design pattern, as it holds all the event handlers for GUI and handles all input from the GUI by the user, making it look like the GUI is doing almost everything.

Command-Subsystems using this design pattern:

The MessageProtocol, MessageTranslator, MessageProtocol and ClientTranslator subsystems all use this design pattern as they build up a large chain of commands and data types including whole objects into a single message protocol to be sent through to different machines.

2.4 Subsystem services

In this section we discuss the services offered by each subsystems followed by the operations of each services. Each operations will show the call they belong to.

Services for UI Handler Control Subsystem:
- Guipopulator
- UserInputCommandService

Services for ClientCache Model Subsystem:
- ClientCacheUpdateNotification

Services for the Client Communication Manager Subsystem:
- ClientTranslator
- ClientConnectionManager

Services for Server Communication Manager Subsystem:
- ServerTranslator
- ServerConnectionManager

Services for Database management subsystem:
- GetNewsFlash
- GetFlashSegment
- GetFlashBoard
- DeleteNewsFlash
- FilterFlashfeed
- DeleteFlashSegment
- PostNewsflash
- CreateFlashSegment
- UploadPhoto
- UpdateProfile

Services	Operation	Description	Class of Operation
Guipopulator	ChangeWindow()	*populates the respective page with its necessary information or alters page due to action*	WindowControl
UserInputCommandService	logout()	*Logs user out, makes sure all required data has persisted, sends garbage cleaner call, sends a call to close connection and makes sure all internal processes are terminated gracefully.*	ProfileWindowHandler
UserInputCommandService	goHome()	*Goes to the FlashFeedWindow*	ProfileWindowHandler
UserInputCommandService	upload()	*Opens a file browser window to select a file*	ProfileWindowHandler
UserInputCommandService	cancel()	*Goes to the FlashFeedWindow*	ProfileWindowHandler
UserInputCommandService	save()	*Save the change information in the text field to the clientCache*	ProfileWindowHandler
UserInputCommandService	run(clientCache* cache)	*Load data from the clientCache to run the ProfileWindow*	ProfileWindowHandler
UserInputCommandService	`run()`	*Load the LoginWindow gui*	LoginWindowHandler
UserInputCommandService	`login()`	*Checks if login was successful*	LoginWindowHandler
UserInputCommandService	`change(bool success)`	*Check if the window change was successful*	LoginWindowHandler
UserInputCommandService	`getProfiles(bool success)`	*check if the profile request was sucessful*	LoginWindowHandler
UserInputCommandService	`getBoardList(bool success)`	*Get a list of flashboard*	LoginWindowHandler
UserInputCommandService	`getPosts(bool success)`	*Gets a newly persisted post ID and asks the user manager to get the newly posted, post from the database and populates it to the view of corresponding subscribed users.*	LoginWindowHandler
UserInputCommandService	change()	*change the flashsegment*	FlashSegmentWindowHandler

UserInputCommandService	segmentExsists(QString segment_name)	*check that a flashsegment exist*	FlashSegmentWindowHandler
UserInputCommandService	itemChanged(QListWidgetItem *current, QListWidgetItem *old)	*items has change*	FlashSegmentWindowHandler
UserInputCommandService	okButtonClicked()	*ok button has been click*	FlashSegmentWindowHandler
UserInputCommandService	addButtonClicked()	*add button has been click*	FlashSegmentWindowHandler
UserInputCommandService	addSegmentReplyReturn(bool updated)	*check for segment notification*	FlashSegmentWindowHandler
UserInputCommandService	addSegmentUpdate()	*update the added segment*	FlashSegmentWindowHandler
UserInputCommandService	deleteButtonClicked()	*delete button was click*	FlashSegmentWindowHandler
UserInputCommandService	deleteSegmentReplyReturn (bool updated)	*notification that segment was delete*	FlashSegmentWindowHandler
UserInputCommandService	deleteSegmentUpdate()	*delete the segment*	FlashSegmentWindowHandler
UserInputCommandService	viewButtonClicked()	*view button was click*	FlashSegmentWindowHandler
UserInputCommandService	formatText(NewsFlash newsflash, bool full_version)	*format the newsflash text*	FlashFeedWindowHandler
UserInputCommandService	loadPosts()	*Gets persisted post IDs and asks for all loaded posts, post from the database and populates it to the view of corresponding subscribed users.*	FlashFeedWindowHandler
UserInputCommandService	readPost(NewsFlash selected_post)	*read the selected newsflash*	FlashFeedWindowHandler
UserInputCommandService	menu(int index)		FlashFeedWindowHandler
UserInputCommandService	postOneSelect()	*select the first newsflash on the flashfeed window*	FlashFeedWindowHandler

UserInputCommandService	`postTwoSelect()`	*select the second newsflash on the flashfeed window*	FlashFeedWindowHandler
UserInputCommandService	`postThreeSelect()`	*select the three newsflash on the flashfeed window*	FlashFeedWindowHandler
UserInputCommandService	`postFourSelect()`	*select the fourth newsflash on the flashfeed window*	FlashFeedWindowHandler
UserInputCommandService	`postFiveSelect()`	*select the fifth newsflash on the flashfeed window*	FlashFeedWindowHandler
UserInputCommandService	`displayPosts(bool success)`	*display the newsflashes*	FlashFeedWindowHandler
UserInputCommandService	`run(clientCache *cache)`	*start the clientCache*	FilterWindowHandler
UserInputCommandService	`change()`	*change the filter setting*	FilterWindowHandler
UserInputCommandService	`allButtonClicked()`	*all button has been click*	FilterWindowHandler
UserInputCommandService	`addButtonClicked()`	*add button has been click*	FilterWindowHandler
UserInputCommandService	`removeButtonClicked()`	*remove button has been click*	FilterWindowHandler
UserInputCommandService	`noneButtonClicked()`	*none button has been click*	FilterWindowHandler
UserInputCommandService	`cancelButtonClicked()`	*cancel button has been click*	FilterWindowHandler
UserInputCommandService	`saveButtonClicked()`	*save button has been click*	FilterWindowHandler
UserInputCommandService	`updatedFilter(bool updated)`	*filter has been updated*	FilterWindowHandler

ClientCacheUpdateNotification	`addSegment(QString course_id, FlashSegment)`	*add a flashsegment*	clientCache
ClientCacheUpdateNotification	`addPostID(QString post_id)`	*add the post_id*	clientCache
ClientCacheUpdateNotification	`removePost(int position)`	*remove the newsflash*	clientCache
ClientCacheUpdateNotification	`removeSegment(QString course_id, QString segment_id )`	*remove the flashsegment*	clientCache
ClientCacheUpdateNotification	`clearPostIDs()`	*clear the post_id(s)*	clientCache
ClientTranslator	`getMessageName(QByteArray message)`	*get the name of the message*	ClientTranslator
ClientTranslator	`getMessageType(QByteArray message)`	*get the type of message*	ClientTranslator
ClientTranslator	`getMessageCategory(QByteArray message)`	*get the category that the message belongs to*	ClientTranslator
ClientTranslator	`isValidXML(QByteArray message)`	*check that a vaild xml was passed*	ClientTranslator
ClientTranslator	`getMessageAttribute(QDomDocument a, QString element, QString attribute)`	*get the attributes of the message*	ClientTranslator
ClientTranslator	`processGetBoardListReply(QByteArray message)`	*process a list of flashboard reply*	ClientTranslator
ClientTranslator	`processGetNewsflashesReply(QByteArray message)`	*process a list of newsflash*	ClientTranslator
ClientTranslator	`processGetProfilesReply(QByteArray message)`	*process a list of profiles*	ClientTranslator
ClientTranslator	`processAuthenticateReply(QByteArray message)`	*process the authentication reply*	ClientTranslator
ClientTranslator	`processGetNewsflashIDsReply(QByteArray message)`	*process the newsflash_id notification*	ClientTranslator

ClientTranslator	`processRequestSucceededReply(QByteArray message)`	*process the request notification*	ClientTranslator
ClientTranslator	`processRequestFailedReply(QByteArray message)`	*process that the request failed*	ClientTranslator
ClientTranslator	`processNewNewsflashUpdate(QByteArray message)`	*process that the newsflash has been update*	ClientTranslator
ClientTranslator	`processNewSegmentUpdate(QByteArray message)`	*process that the flashsegment has been update*	ClientTranslator
ClientTranslator	`processProfileUpdateUpdate(QByteArray message)`	*process that the profile has been updated*	ClientTranslator
ClientTranslator	`processNewsflashDeletedUpdate(QByteArray message)`	*process that the newsflash has been deleted*	ClientTranslator
ClientTranslator	`processSegmentDeletedUpdate(QByteArray message)`	*process that the segment has been deleted*	ClientTranslator
ClientConnectionManager	`setClientCache(clientCache *cache)`	*set the client cache for the user*	CommManager
ClientConnectionManager	`connectTo(QString ip, QString port)`	*the connection of the ip and port*	CommManager
ClientConnectionManager	`login(QString user_id)`	*contains the login as a user_id*	CommManager
ClientConnectionManager	`updateProfile(UserProfile a_profile)`	*update the profile*	CommManager
ClientConnectionManager	`getBoardList()`	*get the list of flashboard*	CommManager
ClientConnectionManager	`getNewsFlashes(QList<QString> newsflashids)`	*get a list of newsflash*	CommManager
ClientConnectionManager	`getProfiles(QList<QString> user_ids)`	*get list of profiles*	CommManager
ClientConnectionManager	`getNewsFlashIDs(QList<QString> flashboards)`	*get a list of newsflash_id for flashboard*	CommManager

ClientConnectionManager	`getNewsFlashIDs(FlashSegment seg)`	*get a list of newsflash_id for flashsegment*	CommManager
ClientConnectionManager	`getNewsFlashIDs(int user_id)`	*get the newsflash_id*	CommManager
ClientConnectionManager	`newNewsFlash(NewsFlash newsflash)`	*contains the new newsflash*	CommManager
ClientConnectionManager	`newSegment(FlashSegment segment)`	*contains a new flashsegment*	CommManager
ClientConnectionManager	`profileUpdate(UserProfile profile)`	*update the profile*	CommManager
ClientConnectionManager	`newsFlashDeleted(QString id)`	*delete the newflash*	CommManager
ClientConnectionManager	`segmentDeleted(QString id)`	*delete the segment*	CommManager
ClientConnectionManager	`updateFilter(QList<FlashBoard> flashboards)`	*update the filter for flashboard*	CommManager
ClientConnectionManager	`loginEmitter(QByteArray message, bool success)`	*emit a signal for login*	CommManager
ClientConnectionManager	`updateProfileEmitter(QByteArray message, bool success)`	*update the emitter for the profile*	CommManager
ClientConnectionManager	`getBoardListEmitter(QByteArray message, bool success)`	*get the list of flashboard emitter*	CommManager
ClientConnectionManager	`getNewsFlashesEmitter(QByteArray message, bool success)`	*get a list of newsflash emitter*	CommManager
ClientConnectionManager	`getProfilesEmitter(QByteArray message, bool success)`	*get a list of profile emitter*	CommManager
ClientConnectionManager	`getNewsFlashIDsEmitter(QByteArray message, bool success)`	*get a list of newsflash_id emitter*	CommManager
ClientConnectionManager	`newNewsFlashEmitter(QByteArray message, bool success)`	*emit a new newsflash*	CommManager

ClientConnectionManager	`newSegmentEmitter(QByteArray message, bool success)`	*emit a new flashsegment*	CommManager
ClientConnectionManager	`profileUpdateEmitter(QByteArray message, bool success)`	*emit that a profile has been updated*	CommManager
ClientConnectionManager	`newsFlashDeletedEmitter(QByteArray message, bool success)`	*emit that a newsflash has been deleted*	CommManager
ClientConnectionManager	`segmentDeletedEmitter(QByteArray message, bool success)`	*emit that a flashsegment has been deleted*	CommManager
ClientConnectionManager	`updateFilterEmitter(QByteArray message, bool success)`	*emit that the filter has been updated*	CommManager
ClientConnectionManager	`disconnected()`	*disconnection from the server machine*	CommManager
ClientConnectionManager	`connected()`	*connect to the server machine*	CommManager
ClientConnectionManager	`incommingMessage()`	*contains the incomming message*	CommManager
ClientConnectionManager	`loginSignal(bool logged_in)`	*check if the login signal was successful*	CommManager
ClientConnectionManager	`updateProfileSignal(bool updated)`	*update the signal of the profile*	CommManager
ClientConnectionManager	`getBoardListSignal(bool updated)`	*check the list of flashboard signal*	CommManager
ClientConnectionManager	`getNewsFlashesSignal(bool updated)`	*check the list of newsflash signal*	CommManager
ClientConnectionManager	`getProfilesSignal(bool updated)`	*check the list of profile signal*	CommManager
ClientConnectionManager	`getPostsSignal(bool updated)`	*check the list of newsflash signal*	CommManager
ClientConnectionManager	`newSegmentSignal(bool updated)`	*check the signal of a new flashsegment*	CommManager

ClientConnectionManager	`newNewsFlashSignal(bool updated)`	*check the signal of a new newsflash signal*	CommManager
ClientConnectionManager	`profileUpdateSignal(bool updated)`	*check the signal of the profile for update*	CommManager
ClientConnectionManager	`newsFlashDeletedSignal(bool updated)`	*check the signal of the newsflash for deletion*	CommManager
ClientConnectionManager	`segmentDeletedSignal(bool updated)`	*check the signal of the flashsegment for deletion*	CommManager
ClientConnectionManager	`updateFilterSignal(bool updated)`	*check the signal of the filter for updates*	CommManager
ClientConnectionManager	`newSegmentReplySignal(bool updated)`	*check the signal for a new flashsegment notification*	CommManager
ClientConnectionManager	`segmentDeletedReplySignal(bool updated)`	*check the signal for the deletion of a flashsegment notification*	CommManager
UpdateProfile	`addUser(int userID, QString avaName, QString name, int accType, QString Photo)`	*Add a user to the database*	addWrapper
PostNewsflash	`addNewsFlashPost(QString newsFlash, int FlashBoard, int userID, int flashSegmentID, int Attachment, QString date)`	*add a newsflash to the database*	addWrapper
FilterFlashfeed	`addFlashBoard(int flashBoardID, QString flashBoardName)`	*add a flashboard to the database*	addWrapper
FilterFlashfeed	`addFilter(int filterID, int userID, int flashBoardID1, int flashBoardID2, int flashBoardID3, int,int)`	*add the filter setting to the database*	addWrapper
CreateFlashSegment	`addFlashSegment(int flashSegmentID,QString flashSegmentName, int flashBoardID)`	*add the flashsegment to the database*	addWrapper
PostNewsflash	`addAttachments(int attachmentID, int newsFlashPostID, QString fileP)`	*add the attachment of a newsflash to the database*	addWrapper
PostNewsflash	`addLinks(int,QString)`	*add link to the database*	addWrapper
FilterFlashfeed	`changeFilter(int,int,int,int,int,int)`	*change the filter setting on the database*	updateWrapper

UpdateProfile	changeAvatar(int,QString)	*edit the avatar name*	updateWrapper
UpdatePhoto	changePhoto(int,QString)	*change the photo in the database*	updateWrapper
DeleteNewsFlash	removeNewsFlashPost(int newsFlashPostID)	*remove a newflash from the database*	removeWrapper
DeleteFlashSegment	removeFlashSegment(int FlashSegmentID)	*remove a flashsegment from the database*	removeWrapper
	removeFlashBoard(int flashBoardID)	*remove a flashboard from the database*	removeWrapper
FilterFlashfeed	removeFilter(int filterID)	*remove the filter settin from the database*	removeWrapper
DeleteNewsFlash	removeAttachment(int attachmentID)	*remove the attachment of a newsflash from the database*	removeWrapper
FilterFlashfeed	getFlashFeed(Filter *courses)	*get the filter setting from the database*	getWrapper
FilterFlashfeed	getNewsFlashFive(Filter *courses, int pointer)	*get five newsflash from the database*	getWrapper
FilterFlashfeed	getFlashFeedSegment(int SegmentID)	*get the filter to show the newflashes in one flashsegment*	getWrapper
FilterFlashfeed	getNewsFlashFiveSegment(int SegmentID , int pointer)	*get five newsflash from a certain setting*	getWrapper
FilterFlashfeed	getFlashFeedUser(int UserID)	*get the flashfeed of a user*	getWrapper
FilterFlashfeed	getNewsFlashFiveUser(int UserID , int pointer)	*get the 5 flashfeed of the user*	getWrapper
GetNewsflash	getNewsFlash(int PostID)	*get a newflash*	getWrapper
GetFlashBoard	getFlashBoard(int flashBoard)	*get a flashboard*	getWrapper

GetFlashSegment	`getSegmentIDs(int flashBoard)`	*get flashsegments contained in a flashboard*	getWrapper
GetNewsflash	`getAttachments(int newsFlashPostID)`	*get the attachment of a newsflash*	getWrapper
UpdateProfile	`getProfile(int userID)`	*get the profile of by user_id*	getWrapper
GetFlashBoard	`getCourseList(int userID)`	*get the list of flashboard by user_id*	getWrapper
FilterFlashfeed	`getNewsFlashIDList(Filter *courses, int max)`	*get the filter setting of a list of newsflash*	getWrapper
FilterFlashfeed	`getNewsFlashIDList(Filter *courses, int range_lte, int range_gte)`	*get a list of newsflash_id*	getWrapper
FilterFlashfeed	`getNewsFlashIDList(Filter *courses, int range_lte, int range_gte, int max)`	*get a list of newsflash_id*	getWrapper
FilterFlashfeed	`getNewsFlashIDList(int user_id,int max)`	*get a list of newsflash_id*	getWrapper
FilterFlashfeed	`getNewsFlashIDList(int user_id,int range_lte, int range_gte)`	*get a list of newsflash_id*	getWrapper
FilterFlashfeed	`getNewsFlashIDList(int user_id,int range_lte, int range_gte, int max)`	*get a list of newsflash_id*	getWrapper
FilterFlashfeed	`getNewsFlashIDList(int segment_id,int flashboard_id,int max,QString a)`	*get a list of newsflash_id*	getWrapper
FilterFlashfeed	`getNewsFlashIDList(int segment_id,int flashboard_id,int range_lte,int range_gte,QString a)`	*get a list of newsflash_id*	getWrapper
FilterFlashfeed	`getNewsFlashIDList(int segment_id,int flashboard_id,int range_lte,int range_gte ,int max,QString a)`	*get a list of newsflash_id*	getWrapper
GetNewsflash	`getLinks(int PostID)`	*get a list of links by post_id*	getWrapper
GetFlashSegment	`getSegment(int segmentID)`	*get a list of flashsegment by segment_id*	getWrapper

GetFlashSegment	`getSegmentsIDs(int flashBoard)`	*get a list of flashsegment by flashboard*	getWrapper
FilterFlashfeed	`getFilter(int user_id)`	*get the filter setting by user_id*	getWrapper
UpdateProfile	`getUserProfPic(int userID)`	*get the user profile picture by user-id*	getWrapper
GetNewsflash	`getAttachmentPointer()`	*get the pointer to an attachment*	getWrapper
ServerConnectionManager	`setSocket(int descriptor)`	*set the socket*	CUClient
ServerConnectionManager	`getUserID()`	*get the user_id*	CUClient
ServerConnectionManager	`connected()`	*create a client connection*	CUClient
ServerConnectionManager	`disconnected()`	*disconnect from the client*	CUClient
ServerConnectionManager	`readyRead()`	*ready to read a message*	CUClient
ServerConnectionManager	`taskResult(QByteArray ba)`	*the result of a task*	CUClient
ServerConnectionManager	`pushMessage(QByteArray buffer)`	*push the message to the client*	CUClient
ServerConnectionManager	`getUserID(int a_user_id)`	*get the user_id*	CUClient
ServerConnectionManager	`startServer()`	*start the server connection*	CUServer
ServerConnectionManager	`pushNewMessage()`	*push a new message from the server*	CUServer
ServerConnectionManager	`result(QByteArray a_message)`	*the result of a message*	CUTask

ServerConnectionManager	`userID(int a_user_id)`	*the user_id as an int*	CUTask
ServerConnectionManager	`run()`	*run the CUTask*	CUTask
CreateDatabase	`createDatabase`	*create the database*	Database
CreateDatabase	`openDatabase()`	*open the database*	Database
CreateDatabase	`closeDatabase()`	*close the database*	Database
CreateDatabase	`createUserTable()`	*create a table for user*	Database
CreateDatabase	`createNewsFlashPostTable()`	*create a table for newflashes*	Database
CreateDatabase	`createFilterTable()`	*create a table for filter*	Database
CreateDatabase	`createAttachmentTable()`	*create a table for attachment*	Database
CreateDatabase	`createFlashBoardTable()`	*create a table for flashboard*	Database
CreateDatabase	`createFlashSegmentTable()`	*create a table for flashsegment*	Database
ServerTranslator	`checkXML(QByteArray message, QByteArray& update, int * a_user_id)`	*check the xml that's being translated*	MessageTranslator
ServerTranslator	`getMessageAttribute(QDomDocument ,QString,QString)`	*get the attribute of a message*	MessageTranslator
ServerTranslator	`getMessageType(QDomDocument)`	*get the type of message*	MessageTranslator
ServerTranslator	`getMessageElement(QDomDocument, QString)`	*get the element of a message*	MessageTranslator

ServerTranslator	`processAuthReq(QByteArray, int *a_user_id)`	*Processes a request for authentication, Log in*	MessageTranslator
ServerTranslator	`processNewsFlashCreation (QByteArray, QByteArray&)`	*process the creation of a newsflash*	MessageTranslator
ServerTranslator	`getRootMessageAttribute( QDomDocument, QString)`	*get the attribut of the root message*	MessageTranslator
ServerTranslator	`processDelFlashSegm(QByteArray, QByteArray&)`	*process the deletion of the flashsegment*	MessageTranslator
ServerTranslator	`processDelNewsFlash(QByteArray, QByteArray&)`	*process the deletion of the newsflash*	MessageTranslator
ServerTranslator	`processCreateSegment(QByteArray, QByteArray&)`	*process the creation of a flashsegment*	MessageTranslator
ServerTranslator	`getMessageElementList(QDomDocument,QString,QString)`	*get the list of message of elements*	MessageTranslator
ServerTranslator	`processGetProfileReq(QByteArray)`	*Processes a request for getting profiles from the client and creates an appropriate reply to be sent back to the client*	MessageTranslator
ServerTranslator	`processGetNewsFlashIDs(QByteArray)`	*process the grabbing newsflash_id*	MessageTranslator
ServerTranslator	`processGetNewsFlash(QByteArray)`	*process the grabbing of a newsflash*	MessageTranslator
ServerTranslator	`getMessageNodeElementList(QDomDocument doc, QString ele)`	*get the list of the MessageNodeElement*	MessageTranslator
ServerTranslator	`getMessageNodeElementListStrings(QDomDocument doc, QString ele)`	*get the list of the MessageNodeElement*	MessageTranslator
ServerTranslator	`processGetBoardList(int)`	*process a list of flashboard*	MessageTranslator
ServerTranslator	`processUpdateProfile(QByteArray, int, QByteArray&)`	*process the updated profile*	MessageTranslator
ServerTranslator	`isValidXML(QDomDocument)`	*contain message is a valid xml*	MessageTranslator

ClientCacheUpdateNotification	`setUserType(QString a_user_type)`	*set the type of user*	clientCache
ClientCacheUpdateNotification	`setUserId(QString a_user_id)`	*set the user id*	clientCache
ClientCacheUpdateNotification	`setCurrentFlashFeedName( QString flash_feed_name)`	*set the name of the current flashfeed*	clientCache
ClientCacheUpdateNotification	`setName(QString mName)`	*set the name of the user*	clientCache
ClientCacheUpdateNotification	`setAvatar(QString an_avatar)`	*set the Avatar(nickname)*	clientCache
ClientCacheUpdateNotification	`setFlashFeedType(int type)`	*set the type of user*	clientCache
ClientCacheUpdateNotification	`setPosition(int position)`	*set the position of which newsflash to show*	clientCache
ClientCacheUpdateNotification	`addNewPost(NewsFlash post)`	*add a newsFlash*	clientCache
ClientCacheUpdateNotification	`setFilter(QList<FlashBoard> course_list)`	*set the filter setting of a user*	clientCache
ClientCacheUpdateNotification	`setCourseList(QList<FlashBoard> mFlashBoards)`	*set the list of flashboard*	clientCache
ClientCacheUpdateNotification	`setPhoto(CUPicture)`	*set the photo*	clientCache
ClientCacheUpdateNotification	`setProfile(UserProfile profile)`	*set the profile*	clientCache
ClientCacheUpdateNotification	`setPosts(QList<NewsFlash> postlist)`	*set the list of newsflash*	clientCache
ClientCacheUpdateNotification	`addMorePosts(QList<NewsFlash> postlist, bool top)`	*add more newsflash*	clientCache
ClientCacheUpdateNotification	`setAuthorIDs(QList<UserProfile> authorlist)`	*set the author_id*	clientCache

ClientCacheUpdateNotification	`getUserId()`	*get the user_id*	clientCache
ClientCacheUpdateNotification	`getFlashFeedType()`	*get the type of flashfeed showing all newsflash, or filter by a single student or filter by course selection setting*	clientCache
ClientCacheUpdateNotification	`getPosition()`	*get the position of the newsflashes*	clientCache
ClientCacheUpdateNotification	`isStudent()`	*check if the user is a student*	clientCache
ClientCacheUpdateNotification	`isEmpty()`	*check if the client cache is empty*	clientCache
ClientCacheUpdateNotification	`getCurrentFlashFeedName()`	*get the name of the current flashfeed*	clientCache
ClientCacheUpdateNotification	`getName()`	*get the Name of the user*	clientCache
ClientCacheUpdateNotification	`getAvatar()`	*get the avatar(nickname) of the user*	clientCache
ClientCacheUpdateNotification	`getCourseID(QString course_name)`	*get the course_id*	clientCache
ClientCacheUpdateNotification	`getPost(int)`	*Gets a newly persisted post ID and asks the user manager to get the newly posted, post from the database and populates it to the view of corresponding subscribed users.*	clientCache
ClientCacheUpdateNotification	`getPhoto()`	*get the photo in the clientCache*	clientCache
ClientCacheUpdateNotification	`getCourseList()`	*get a list of flashboards*	clientCache
ClientCacheUpdateNotification	`getSegmentList(QString segment_id)`	*get a list of flashsegment*	clientCache
ClientCacheUpdateNotification	`getPostIDs(int j, int k)`	*get post ids*	clientCache
ClientCacheUpdateNotification	`getProfileList()`	*get a list of profile*	clientCache
ClientCacheUpdateNotification	`getCommManager()`	*get the CommManager class*	clientCache

2.5 Message Protocol

Message protocol has already been decided. We are building our project based on the message protocol.

3 Object Design

3.1 Overview

UI Handler Control Service

LoginWindowHandler- handles input and output to the LoginWindow, and performs tasks based on the information taken.

FilterWindowHandler- handles input and output to the FilterWindow, and performs tasks based on the information taken.

FlashSegmentWindowHandler- handles input and output to the FlashSegmentWindow, and performs tasks based on the information taken.

FlashFeedWindowHandler- handles input and output to the FlashFeedWindow, and performs tasks based on the information taken.

ProfileWIndowHandler- handles input and output to the ProfileWIndow, and performs tasks based on the information taken.

WindowControl - keeps track of all of the Ui pages and changes them to the appropriate page as necessary.

Client Cache Model Service

clientCache- stores all of the client data of a session for easy re access without calling the database.

Client Communication Service

ClientTranslator- translate all incoming message and process to the client.

CommManager- allows the Client to talk with the server.

MessageProtocol - class allows the client to communicated with the server creating objects.

MessageTranslator- class translate incoming information from the client to the server.

Server Communication Service

MessageProtocol - class allows the client to communicated with the server creating objects.

MessageTranslator- this class translator receives and parses XML files

from the server and forms a reply that the client will be able to decipher.

CUClient- class creates sockets and connects to the server. Controls socket connection flow to not allow operations done by one client to overlap or interfere with other processes.

CUServer- class starts the server service and listens for client connections through the sockets.

CUTask- class that contains the messages in byteArray.

Database Management service

addWrapper- wrapper class to indirectly access SQL based function add new entries into model tables.

updatewrapper- wrapper class to indirectly access SQL based function to update entries from the model tables.

removeWrapper- wrapper class to indirectly access SQL based function to remove entries from the model tables.

getWrapper- wrapper class to indirectly access SQL based function to get entries from the model tables.

SQLApi- bridges all data that needs to be persisted in the database and all data being taken from the database. It gets all information then maps to or from the database and sends it to its next step, i.e. database up to user or to a manager.

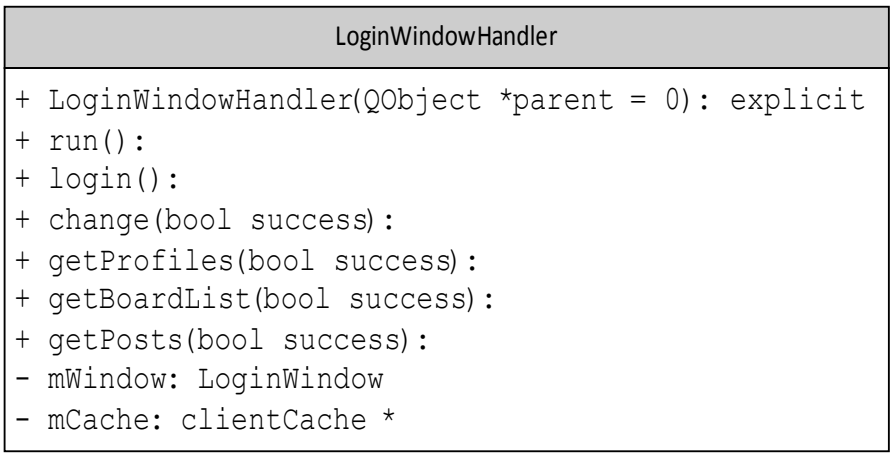
3.2 Class Interfaces

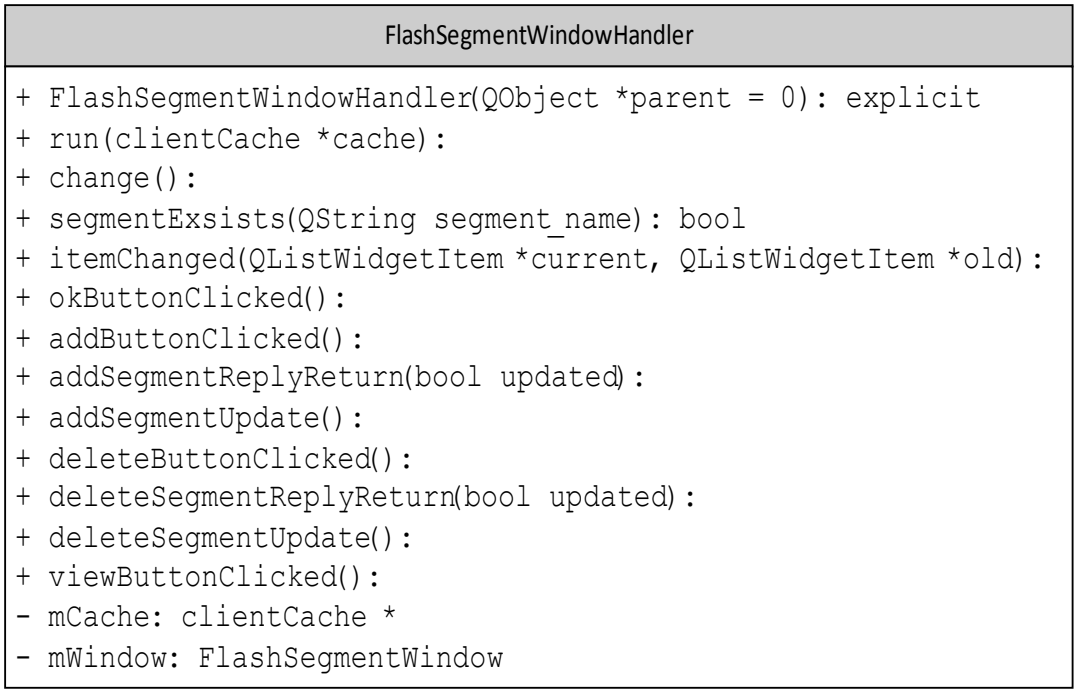
On the following pages you will see the class break down based on the subsystems to which they belong, in the goal of space management all newsFlashPosts object that will be used as parameters have been substituted for Post.

UI Handler Control Subsystem

WindowControl
+ changeWindow(): static + changeWindow(int, clientCache*): static + PROFILE_PAGE: static const int + FLASHFEED_PAGE: static const int + FLASHSEGMENT_PAGE: static const int + FILTER_PAGE: static const int

ProfileWindowHandler
+ ProfileWindowHandler(QObject *parent = 0): explicit + run(clientCache* cache): + firstTimeSaved(): bool + errorBox(): + goHome(): + logout(): + upload(): + cancel(): + save(): + change(bool recieved): - mCache: clientCache * - mWindow: ProfileWindow - uploaded_photo: QImage - ext: QString


LoginWindowHandler
+ LoginWindowHandler(QObject *parent = 0): explicit
+ run():
+ login():
+ change(bool success):
+ getProfiles(bool success):
+ getBoardList(bool success):
+ getPosts(bool success):
- mWindow: LoginWindow
- mCache: clientCache *


FlashSegmentWindowHandler
+ FlashSegmentWindowHandler(QObject *parent = 0): explicit
+ run(clientCache *cache):
+ change():
+ segmentExsists(QString segment_name): bool
+ itemChanged(QListWidgetItem *current, QListWidgetItem *old):
+ okButtonClicked():
+ addButtonClicked():
+ addSegmentReplyReturn(bool updated):
+ addSegmentUpdate():
+ deleteButtonClicked():
+ deleteSegmentReplyReturn(bool updated):
+ deleteSegmentUpdate():
+ viewButtonClicked():
- mCache: clientCache *
- mWindow: FlashSegmentWindow

FlashFeedWindowHandler
+ FlashFeedWindowHandler(QObject *parent = 0): explicit + run(clientCache *cache): + formatText(NewsFlash newsflash, bool full_version): QString + loadPosts(): + readPost(NewsFlash selected_post): + goHome(): + logout(): + menu(int index): + postOneSelect(): + postTwoSelect(): + postThreeSelect(): + postFourSelect(): + postFiveSelect(): + displayPosts(bool success): - mCache: clientCache * - mWindow: FlashFeedWindow - scene: QGraphicsScene *

FilterWindowHander
+ FilterWindowHandler(QObject *parent = 0): explicit + run(clientCache *cache): + change(): + allButtonClicked(): + addButtonClicked(): + removeButtonClicked(): + noneButtonClicked(): + cancelButtonClicked(): + saveButtonClicked(): + updatedFilter(bool updated): - mCache: clientCache * - mWindow: FilterWindow - mFilter: QList<FlashBoard>

Client Communication Manager Subsystem

CommManager
+ CommManager(QObject *parent = 0): explicit + setClientCache(clientCache *cache): + connectTo(QString ip, QString port): + login(QString user_id): + updateProfile(UserProfile a_profile): + getBoardList(): + getNewsFlashes(QList<QString> newsflashids): + getProfiles(QList<QString> user_ids): + getNewsFlashIDs(QList<QString> flashboards): + getNewsFlashIDs(FlashSegment seg): + getNewsFlashIDs(int user_id): + newNewsFlash(NewsFlash newsflash): + newSegment(FlashSegment segment): + profileUpdate(UserProfile profile): + newsFlashDeleted(QString id): + segmentDeleted(QString id): + updateFilter(QList<FlashBoard> flashboards): + loginEmitter(QByteArray message, bool success): + updateProfileEmitter(QByteArray message, bool success): + getBoardListEmitter(QByteArray message, bool success): + getNewsFlashesEmitter(QByteArray message, bool success): + getProfilesEmitter(QByteArray message, bool success): + getNewsFlashIDsEmitter(QByteArray message, bool success): + newNewsFlashEmitter(QByteArray message, bool success): + newSegmentEmitter(QByteArray message, bool success): + profileUpdateEmitter(QByteArray message, bool success): + newsFlashDeletedEmitter(QByteArray message, bool success): + segmentDeletedEmitter(QByteArray message, bool success): + updateFilterEmitter(QByteArray message, bool success): + disconnected(): + connected(): + incommingMessage(): + loginSignal(bool logged_in): + updateProfileSignal(bool updated): + getBoardListSignal(bool updated): + getNewsFlashesSignal(bool updated): + getProfilesSignal(bool updated): + getPostsSignal(bool updated): + newSegmentSignal(bool updated): + newNewsFlashSignal(bool updated): + profileUpdateSignal(bool updated): + newsFlashDeletedSignal(bool updated): + segmentDeletedSignal(bool updated): + updateFilterSignal(bool updated): + newSegmentReplySignal(bool updated): + segmentDeletedReplySignal(bool updated): - sock: QTcpSocket * - mCache: clientCache * - ip, port: QString - last_message_types: QList<QString> - message: QByteArray

ClientTranslator
+ getMessageName(QByteArray message) : static QString + getMessageType(QByteArray message) : static QString + getMessageCategory(QByteArray message) : static QString + isValidXML(QByteArray message) : static bool + getMessageAttribute(QDomDocument a, QString element, QString attribute) : static QString + processGetBoardListReply(QByteArray message) : static QList<FlashBoard> + processGetNewsflashesReply(QByteArray message) : static QList<NewsFlash> + processGetProfilesReply(QByteArray message) : static QList<UserProfile> + processAuthenticateReply(QByteArray message) : static QString + processGetNewsflashIDsReply(QByteArray message) : static QList<QString> + processRequestSucceededReply(QByteArray message) : static int + processRequestFailedReply(QByteArray message) : static QString + processNewNewsflashUpdate(QByteArray message) : static NewsFlash + processNewSegmentUpdate(QByteArray message) : static FlashSegment + processProfileUpdateUpdate(QByteArray message) : static UserProfile + processNewsflashDeletedUpdate(QByteArray message) : static int + processSegmentDeletedUpdate(QByteArray message) : static int

Client Cache Model Subsystem

clientCache
+ setup(): + getUserId(): int + getFlashFeedType(): int + getPosition(): int + isStudent(): bool + isEmpty(): bool + getCurrentFlashFeedName(): QString + getName(): QString + getAvatar(): QString + getCourseID(QString course_name): QString + getPost(int): NewsFlash + getPhoto(): CUPicture + getCourseList(): QList<FlashBoard> + getSegmentList(QString segment_id): QList<FlashSegment> + getPostIDs(int j, int k): QList<QString> + getProfileList(): QList<UserProfile> + getCommManager(): CommManager* + setUserType(QString a_user_type): + setUserId(QString a_user_id): + setCurrentFlashFeedName(QString flash_feed_name): + setName(QString mName): + setAvatar(QString an_avatar): + setFlashFeedType(int type): + setPosition(int position): + addNewPost(NewsFlash post): + setFilter(QList<FlashBoard> course_list): + setCourseList(QList<FlashBoard> mFlashBoards): + setPhoto(CUPicture): + setProfile(UserProfile profile): + setPosts(QList<NewsFlash> postlist): + addMorePosts(QList<NewsFlash> postlist, bool top): + setAuthorIDs(QList<UserProfile> authorlist): + addSegment(QString course_id, FlashSegment): bool + addPostID(QString post_id): + removePost(int position): + removeSegment(QString course_id, QString segment_id): + clearPostIDs(): - userID: int - student: bool - flashFeedType: int - position: int - currentFlashFeedName: QString - profilePicture: CUPicture - name: QString - avatar: QString - author_list: QList<UserProfile> - posts: QList<NewsFlash> - post_ids: QList<QString> - flashBoards: QList<FlashBoard> - mComm: CommManager *

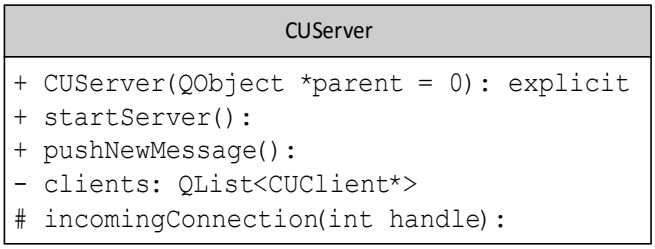
Server Communication Manager Subsystem

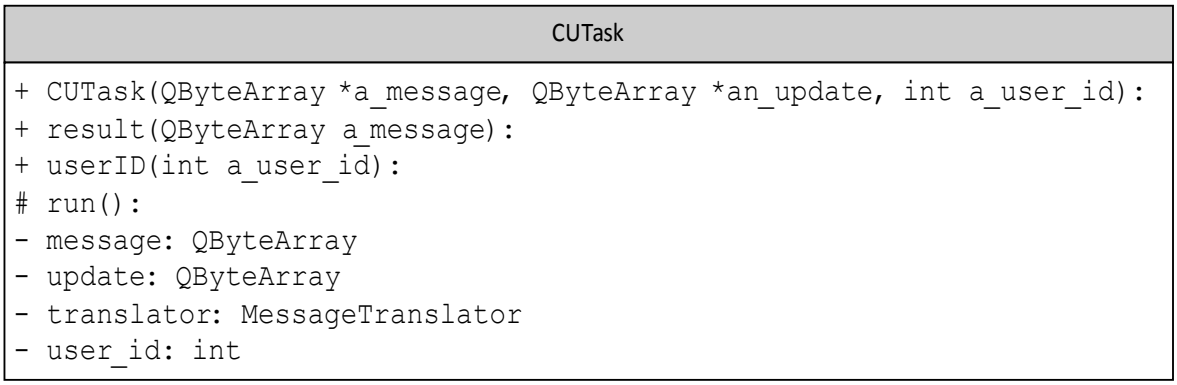
MessageTranslator
+ api: SQLApi * + checkXML(QByteArray message, QByteArray& update, int * a_user_id): QByteArray + getMessageAttribute(QDomDocument ,QString,QString): QString + getMessageType(QDomDocument): QString + getMessageElement(QDomDocument, QString): QString + processAuthReq(QByteArray, int *a_user_id): QByteArray + processNewsFlashCreation(QByteArray, QByteArray&): QByteArray + getRootMessageAttribute(QDomDocument, QString): QString + processDelFlashSegm(QByteArray, QByteArray&): QByteArray + processDelNewsFlash(QByteArray, QByteArray&): QByteArray + processCreateSegment(QByteArray, QByteArray&): QByteArray + getMessageElementList(QDomDocument,QString,QString): QList<QString> + processGetProfileReq(QByteArray): QByteArray + processGetNewsFlashIDs(QByteArray): QByteArray + processGetNewsFlash(QByteArray): QByteArray + getMessageNodeElementList(QDomDocument doc, QString ele): QList<QByteArray> + getMessageNodeElementListStrings(QDomDocument doc, QString ele): QList<QString> + processGetBoardList(int): QByteArray + processUpdateProfile(QByteArray, int, QByteArray&): QByteArray + isValidXML(QDomDocument): bool + userID: int

CUClient
+ CUClient(QObject *parent = 0): explicit + setSocket(int descriptor): + getUserID(): int + connected(): + disconnected(): + readyRead(): + taskResult(QByteArray ba): + pushMessage(QByteArray buffer): + getUserID(int a_user_id): - socket: QTcpSocket * - user_id: int - translator: MessageTranslator

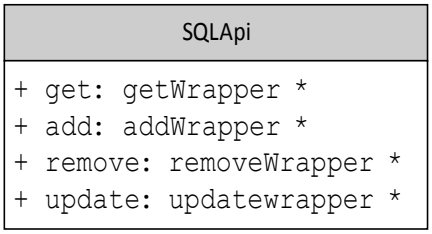
MessageProtocol

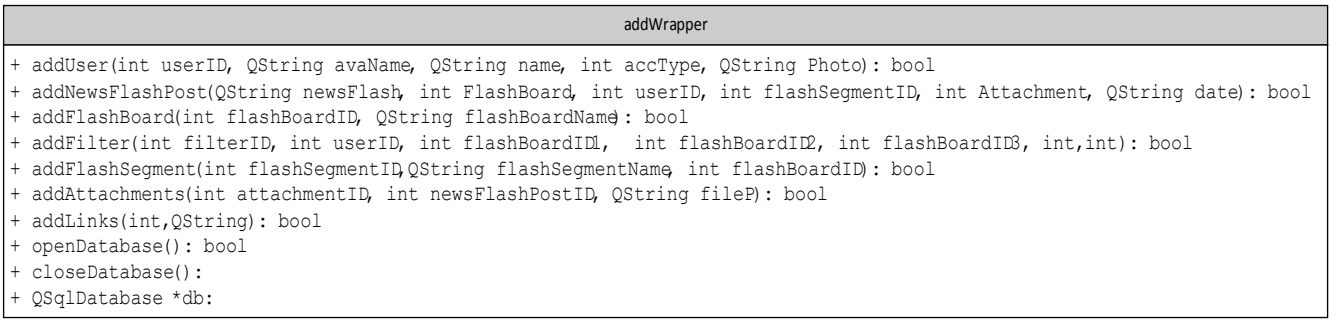
+ REPLY: static const QString
+ UPDATE: static const QString
+ REQUEST: static const QString
+ AUTHENTICATE: static const QString
+ UNAUTHENTICATE: static const QString
+ GET_BOARD_LIST: static const QString
+ GET_NEWSFLASHES: static const QString
+ GET_PROFILES: static const QString
+ GET_NEWSFLASH_IDS: static const QString
+ FILTER_BOARD: static const QString
+ NEW_NEWSFLASH: static const QString
+ NEW_SEGMENT: static const QString
+ CREATE_NEWSFLASH: static const QString
+ CREATE_SEGMENT: static const QString
+ DELETE_NEWSFLASH: static const QString
+ DELETE_SEGMENT: static const QString
+ NEWSFLASH_DELETED: static const QString
+ SEGMENT_DELETED: static const QString
+ UPDATE_PROFILE: static const QString
+ PROFILE_UPDATE: static const QString
+ REQUEST_FAILED: static const QString
+ REQUEST_SUCCEEDED: static const QString
+ createAuthenticateRequest(QString a_userid, QString a_password): static QByteArray
+ createUnauthenticateRequest(): static QByteArray
+ createGetNewsFlashIDsRequest(QString userid): static QByteArray
+ createGetNewsFlashIDsRequest(): static QByteArray
+ createGetNewsFlashIDsRequest(QList<QString> flashboardlist): static QByteArray
+ createGetBoardListRequest(): static QByteArray
+ createGetNewsflashesRequest(QList<QString> newsflashlist): static QByteArray
+ createGetProfilesRequest(QList<QString> userlist): static QByteArray
+ createCreateNewsflashRequest(NewsFlash a_newsflash): static QByteArray
+ createCreateSegmentRequest(FlashSegment a_flashsegment): static QByteArray
+ createUpdateProfileRequest(CUPicture a_picture): static QByteArray
+ createUpdateProfileRequest(QString a_nickname): static QByteArray
+ createFilterBoardRequest(QList<FlashBoard> flashboards): static QByteArray
+ createDeleteNewsflashRequest(QList<NewsFlash> newsflashes): static QByteArray
+ createDeleteSegmentRequest(QList<FlashSegment> segments): static QByteArray
+ createGetBoardListReply(QList<FlashBoard> flashboards): static QByteArray
+ createGetNewsflashesReply(QList<NewsFlash> newsflashes): static QByteArray
+ createGetProfilesReply(QList<UserProfile> profiles): static QByteArray
+ createAuthenticateReply(UserProfile profile): static QByteArray
+ createGetNewsflashIDsReply(QList<NewsFlash> newsflashes): static QByteArray
+ createRequestSucceededReply(int id): static QByteArray
+ createRequestFailedReply(QString exception_code): static QByteArray
+ createNewNewsflashUpdate(NewsFlash newsflash): static QByteArray
+ createNewSegmentUpdate(FlashSegment segment): static QByteArray
+ createProfileUpdateUpdate(UserProfile profile): static QByteArray
+ createNewsflashDeletedUpdate(int id): static QByteArray
+ createSegmentDeletedUpdate(int id): static QByteArray
- addXMLVersion(QDomDocument &message): static void
- createRequest(QDomDocument &message, QString a_category, QString a_type): static QDomElement
- finalizeRequest(QDomDocument &message, QDomElement request): static QByteArray
- createReply(QDomDocument &message, QString a_category, QString a_type): static QDomElement
- finalizeReply(QDomDocument &message, QDomElement reply): static QByteArray
- createUpdate(QDomDocument &message, QString a_category, QString a_type): static QDomElement
- finalizeUpdate(QDomDocument &message, QDomElement update): static QByteArray
- appendPicture(QDomDocument &message, QDomElement &request, CUPicture a_picture): static
- appendURL(QDomDocument &message, QDomElement &request, CUUrl a_url): static
- appendNewsFlash(QDomDocument &message, QDomElement &request, NewsFlash a_newsflash): static
- appendFlashSegment(QDomDocument &message, QDomElement &request, FlashSegment a_flashsegment): static
- appendUserProfile(QDomDocument &message, QDomElement &request, UserProfile a_userprofile): static
- appendFlashBoard(QDomDocument &message, QDomElement &request, FlashBoard a_flashboard): static

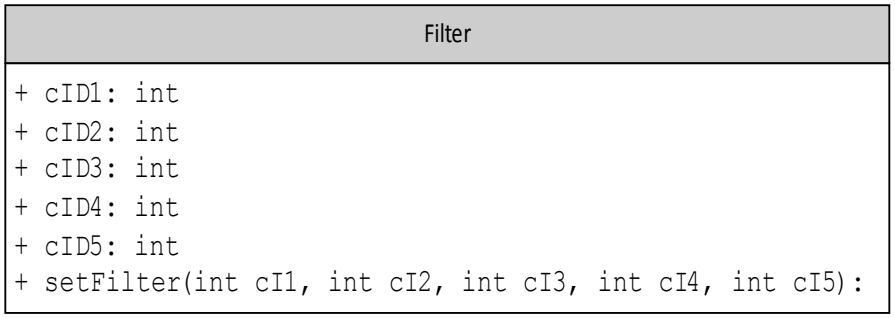

CUServer
+ CUServer(QObject *parent = 0): explicit
+ startServer():
+ pushNewMessage():
- clients: QList<CUClient*>
incomingConnection(int handle):

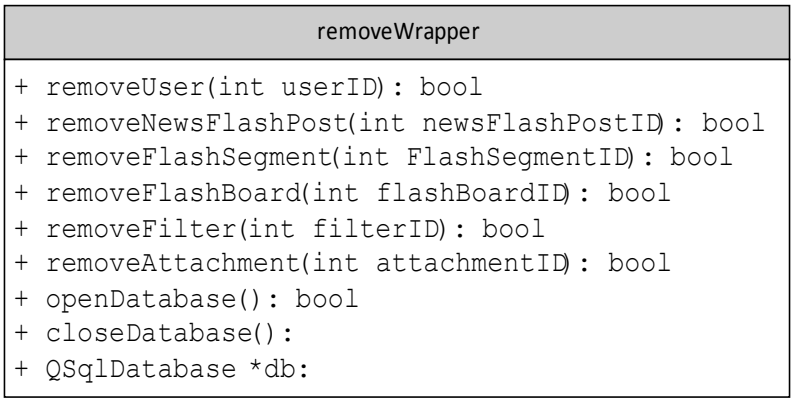

CUTask
+ CUTask(QByteArray *a_message, QByteArray *an_update, int a_user_id):
+ result(QByteArray a_message):
+ userID(int a_user_id):
# run():
- message: QByteArray
- update: QByteArray
- translator: MessageTranslator
- user_id: int

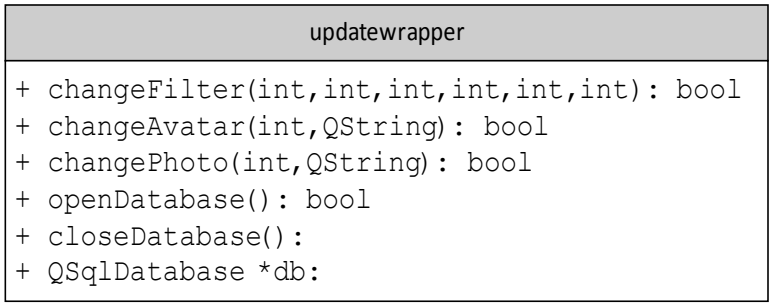
Database Management Subsystem

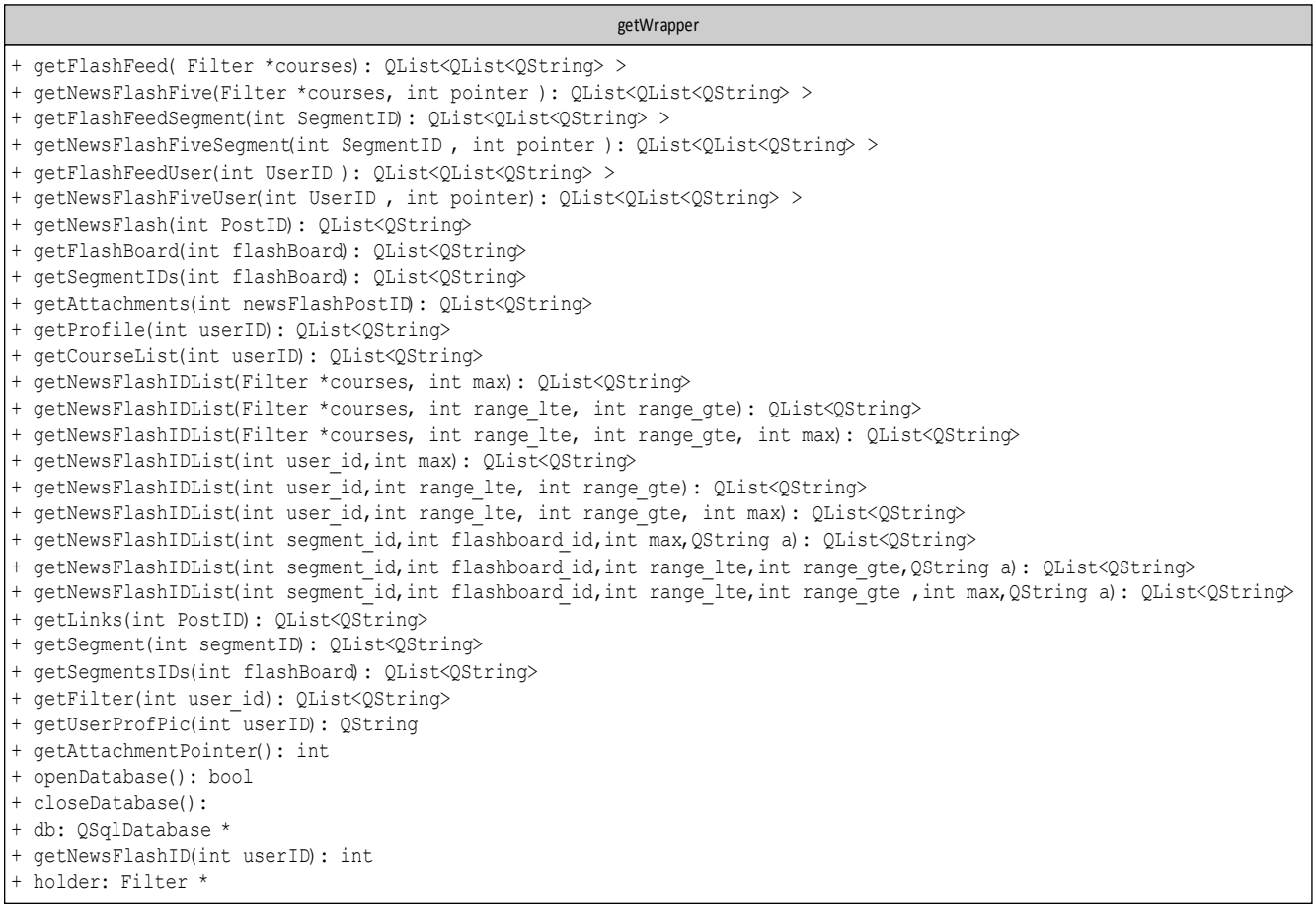


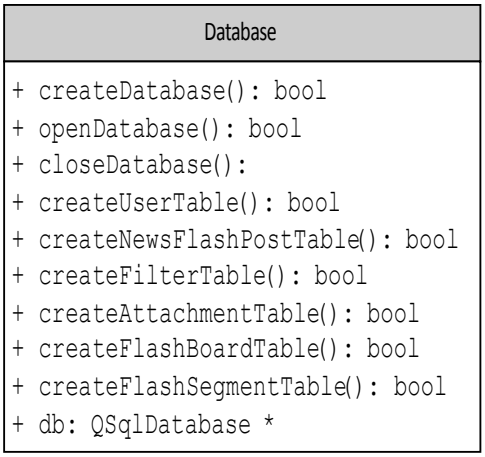




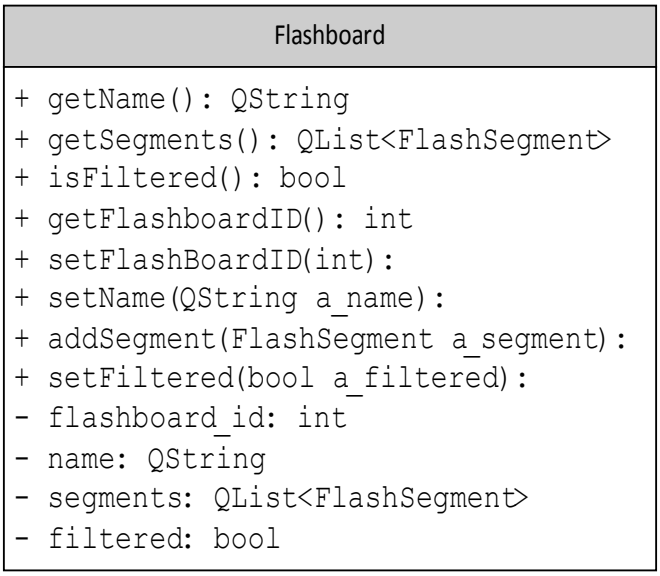



updatewrapper
+ changeFilter(int,int,int,int,int,int): bool
+ changeAvatar(int,QString): bool
+ changePhoto(int,QString): bool
+ openDatabase(): bool
+ closeDatabase():
+ QSqlDatabase *db:


getWrapper
+ getFlashFeed(Filter *courses): QList<QList<QString> >
+ getNewsFlashFive(Filter *courses, int pointer): QList<QList<QString> >
+ getFlashFeedSegment(int SegmentID): QList<QList<QString> >
+ getNewsFlashFiveSegment(int SegmentID , int pointer): QList<QList<QString> >
+ getFlashFeedUser(int UserID): QList<QList<QString> >
+ getNewsFlashFiveUser(int UserID , int pointer): QList<QList<QString> >
+ getNewsFlash(int PostID): QList<QString>
+ getFlashBoard(int flashBoard): QList<QString>
+ getSegmentIDs(int flashBoard): QList<QString>
+ getAttachments(int newsFlashPostID): QList<QString>
+ getProfile(int userID): QList<QString>
+ getCourseList(int userID): QList<QString>
+ getNewsFlashIDList(Filter *courses, int max): QList<QString>
+ getNewsFlashIDList(Filter *courses, int range_lte, int range_gte): QList<QString>
+ getNewsFlashIDList(Filter *courses, int range_lte, int range_gte, int max): QList<QString>
+ getNewsFlashIDList(int user_id,int max): QList<QString>
+ getNewsFlashIDList(int user_id,int range_lte, int range_gte): QList<QString>
+ getNewsFlashIDList(int user_id,int range_lte, int range_gte, int max): QList<QString>
+ getNewsFlashIDList(int segment_id,int flashboard_id,int max,QString a): QList<QString>
+ getNewsFlashIDList(int segment_id,int flashboard_id,int range_lte,int range_gte,QString a): QList<QString>
+ getNewsFlashIDList(int segment_id,int flashboard_id,int range_lte,int range_gte ,int max,QString a): QList<QString>
+ getLinks(int PostID): QList<QString>
+ getSegment(int segmentID): QList<QString>
+ getSegmentsIDs(int flashBoard): QList<QString>
+ getFilter(int user_id): QList<QString>
+ getUserProfPic(int userID): QString
+ getAttachmentPointer(): int
+ openDatabase(): bool
+ closeDatabase():
+ db: QSqlDatabase *
+ getNewsFlashID(int userID): int
+ holder: Filter *

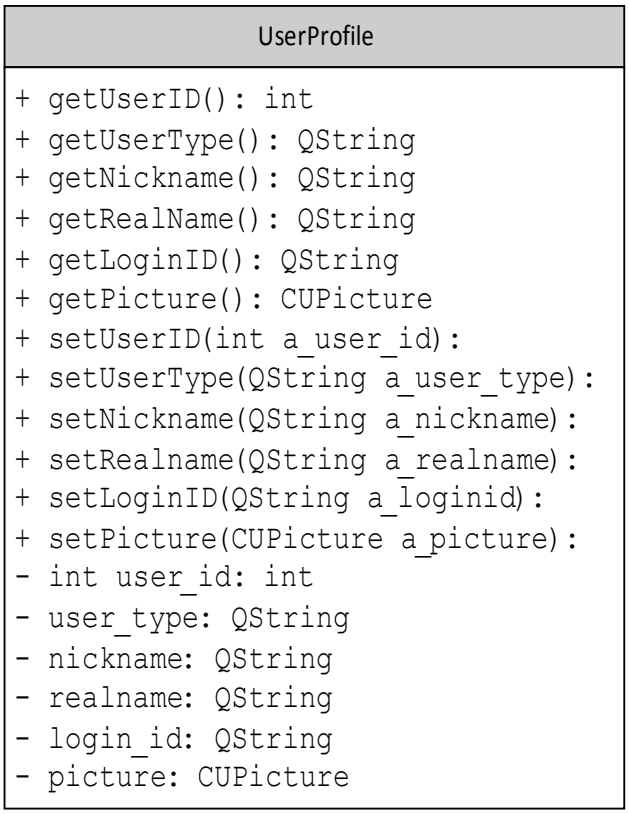


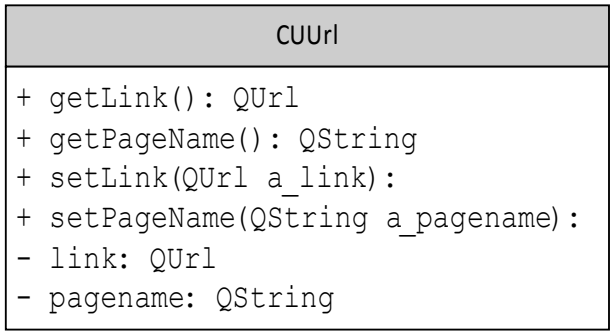
Entity Objects

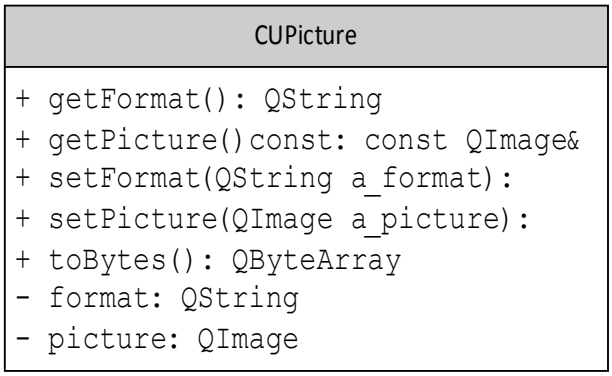


Flashsegment
+ getSegmentID(): int + getFlashboardID(): int + getName(): QString + setSegmentID(int a_segment_id): + setFlashboardID(int a_flashboard_id): + setName(QString a_name): - name: QString - segment_id, flashboard_id: int

Newsflash
+ NewsFlash(int a_newsflash_id, int a_author_id, int a_flashboard_id, int a_segment_id): + NewsFlash(int a_newsflash_id, int a_author_id, int a_flashboard_id, int a_segment_id, QString post_text): + NewsFlash(int a_newsflash_id, int a_author_id, int a_flashboard_id, int a_segment_id, QString post_text, QList<CUPicture> pics): + NewsFlash(int a_newsflash_id, int a_author_id, int a_flashboard_id, int a_segment_id, QString post_text, QList<CUUrl> links): + NewsFlash(int a_newsflash_id, int a_author_id, int a_flashboard_id, int a_segment_id, QString post_text, QList<CUPicture> pics, QList<CUUrl> links): + getNewsFlashID(): int + getAuthorID(): int + getFlashBoardID(): int + getSegmentID(): int + getPictures(): QList<CUPicture> + getURLs(): QList<CUUrl> + getText(): QString + getAuthorTag(): QString + setNewsFlashID(int a_newsflashid); + setAuthorID(int an_authorid); + setFlashBoardID(int a_flashboardid); + setSegmentID(int a_segmentid); + setAuthorTag(QString a_tag); + addPicture(CUPicture a_picture); + addURL(CUUrl a_url); - newsflash_id: int - author_id: int - flashboard_id: int - segment_id: int - text: QString - author_tag: QString - pictures: QList<CUPicture> - urls: QList<CUUrl>


UserProfile
+ getUserID(): int
+ getUserType(): QString
+ getNickname(): QString
+ getRealName(): QString
+ getLoginID(): QString
+ getPicture(): CUPicture
+ setUserID(int a_user_id):
+ setUserType(QString a_user_type):
+ setNickname(QString a_nickname):
+ setRealname(QString a_realname):
+ setLoginID(QString a_loginid):
+ setPicture(CUPicture a_picture):
- int user_id: int
- user_type: QString
- nickname: QString
- realname: QString
- login_id: QString
- picture: CUPicture


CUUrl
+ getLink(): QUrl
+ getPageName(): QString
+ setLink(QUrl a_link):
+ setPageName(QString a_pagename):
- link: QUrl
- pagename: QString


CUPicture
+ getFormat(): QString
+ getPicture()const: const QImage&
+ setFormat(QString a_format):
+ setPicture(QImage a_picture):
+ toBytes(): QByteArray
- format: QString
- picture: QImage