Term Project: *ChatApp*

Design Document

Table of Contents

1 Introduction 2

1.1 Purpose and Scope 2

1.2 Target Audience 2

1.3 Terms and Definitions 2

2 Design Considerations 3

2.1 Constraints and Dependencies 3

2.2 Methodology 3

3 System Overview 4

4 System Architecture 6

4.1 Register 7

4.2 Log In 8

4.3 Chat 9

4.3.1 Private Chat 9

4.3.2 Public Chat 10

4.4 Access Chat Records 11

4.5 Log Out 12

5 Detailed System Design 13

5.1 Register 13

5.2 Log In 15

5.3 Chat 16

5.3.1 Private Chat 16

5.3.2 Public Chat 17

5.4 Access Chat Records 18

5.5 Log Out 19

# Introduction

## Purpose and Scope

The purpose of this document is to describe the implementation of the ChatApp described in the ChatApp requirements documentation.

## Target Audience

The target audience for this document are Professor Xie and Bin Lin.

## Terms and Definitions

|  |  |
| --- | --- |
| Chat Environment | an area where messages received are displayed for other chatters. |
| Records Environment | an area where previous posts are printed for chatters to view. |
| Chatter 1 | a user of the application. |
| Chatter 2 | one other user. |
| Chatter N | one or more other users. |
| ID | user name chatter wants to be identified as. |
| Registry | database of accounts |

# Design Considerations

Describe the purpose of this section and provide an outline of its subsections. Only a few sentences are expected here.

The purpose of this section is to list the limitations of the application and describe how I came to the design described later in this document.

## Constraints and Dependencies

Functional Requirements:

The functional requirements for this application are that a user must be able to log in, log out, chat publicly with other users, chat privately with individual users, and access their chat history.

Non-Functional Requirements:

The non-functional requirements are that the application must be implemented using the Java programming language and must be scalable to a minimum of one hundred people.

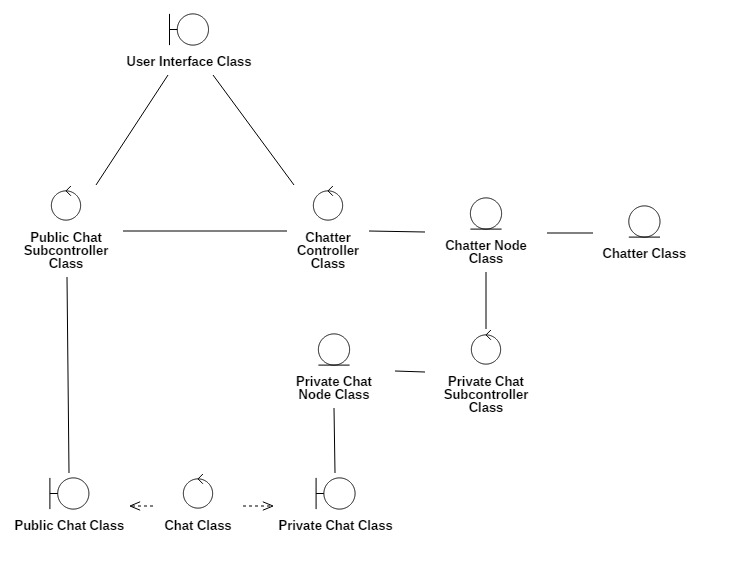
## Methodology

I used the object-oriented design methodology. I broke up all the requirements into smaller separate pieces and incrementally updated and improved them until I had a program that works.

# System Overview

This application will be used to allow people to communicate with all other registered people logged in to the app as well as having private communications with another chatter. Chatters will also be able to search their message history. The public chat sub-controller will be accessible to all users and will contain a public chat object which is a subclass of the chat class. The chatter controller class will be responsible for maintaining the data structure of the registered users. The registry will be implemented as a binary search tree data structure where each node is a chatter. The public chat sub-controller and the chatter controller class will communicate with each other when a chatter has entered the public chat environment. Each chatter node will contain a chatter object which holds the user id and password. The node will also contain the private chat sub-controller class which manages all the private chats the chatter has made. Inside each sub-controller class will be another binary search tree data structure of private chat objects for each chat they have with other chatters. Each node in this tree will contain a private chat class which is, like the public chat class, a subclass of chat class. Chat class will hold the thread of messages for both the one public chat class that will be managed and the n x m private chats that will be made (n = # of chatters, m = # of private chats). Refer to Figure 3.1 and 4.1.

**Class Diagram:**

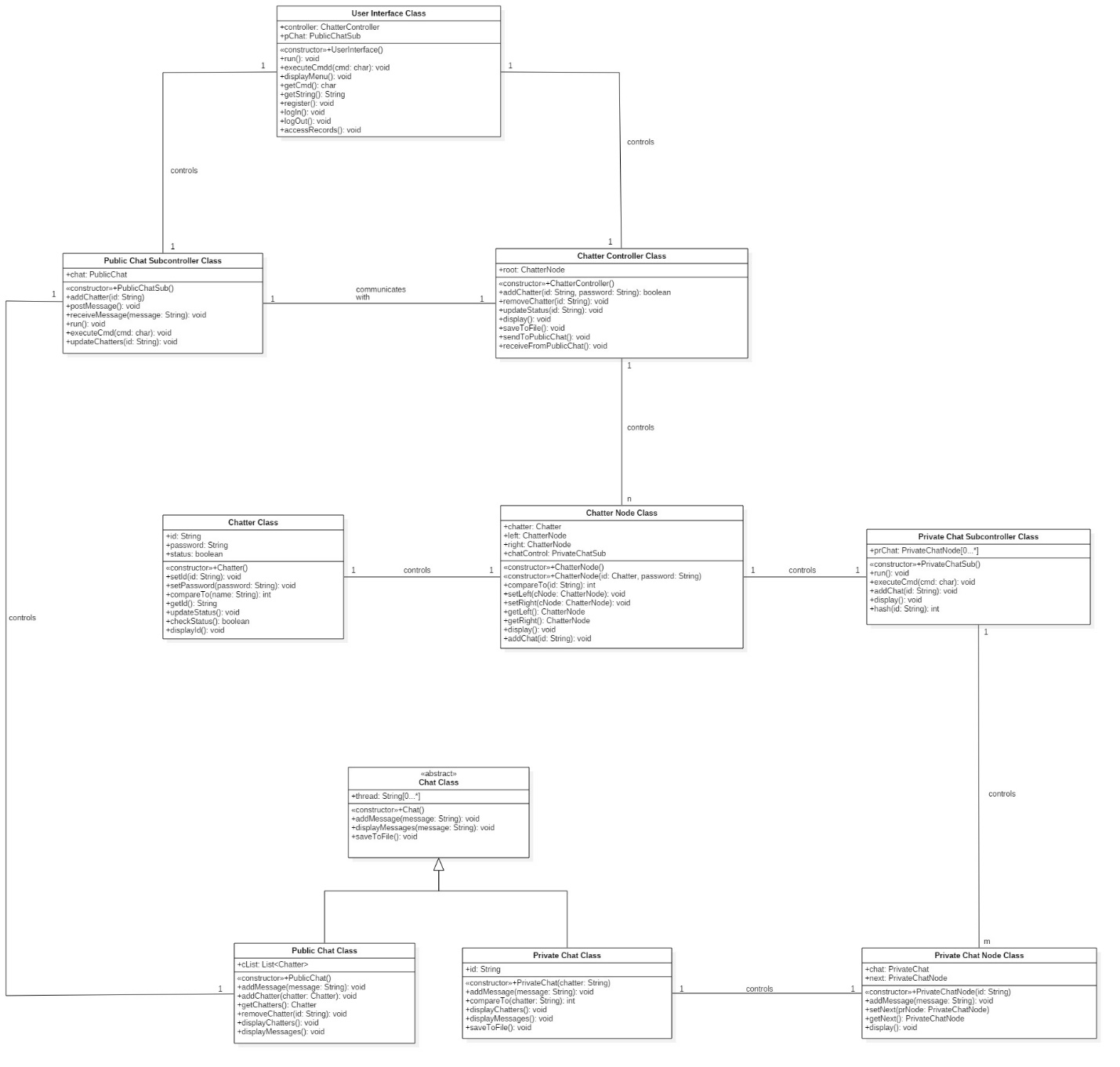


**Figure 3.1**

# System Architecture

There are five required subsystems to this program: register, log in, chat, access chat records, and log out. Chat is broken up into a public chat, with other logged in chatters, and a private chat, with two chatters.

**Detailed Class Diagram:**

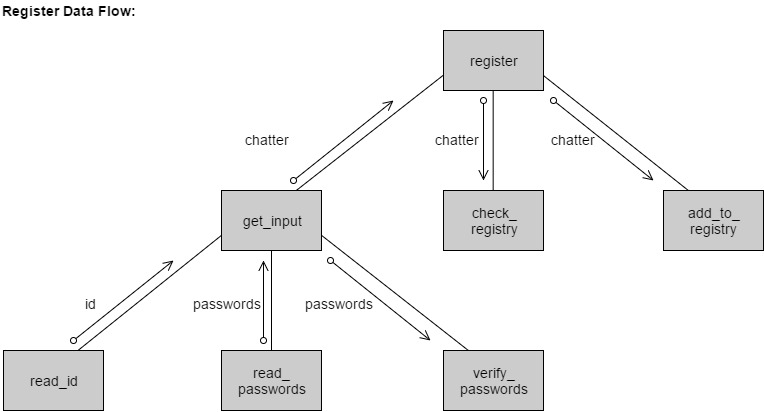


**Figure 4.1**

## Register

Provide a detailed, high level description of the first subsystem, component or object of your system. Your choice of terms should reflect the development methodology you chose in the design considerations section.

You should provide a detailed description of the interface. A high-level description of the internal architecture is also appropriate here. You may want to outline any important algorithms as well.

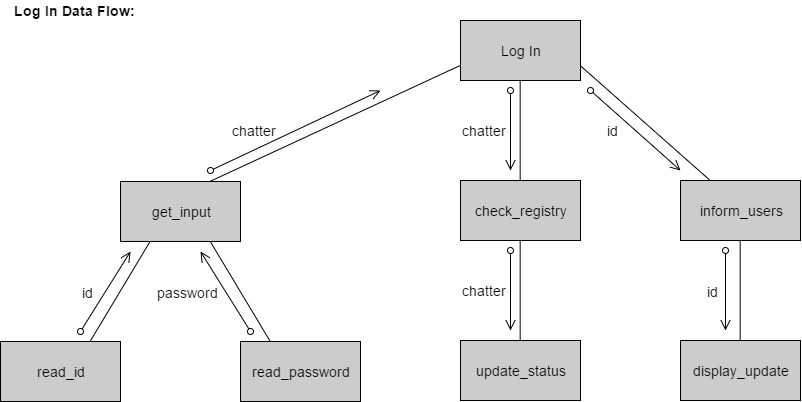


## Figure 4.2

## The interface for register will consist of one page. It will be accessible through the main page described in the next section. Three text boxes will be in the middle of the page, one underneath another. The first text box will be for the chatter to enter their id. The next two text boxes will be for the chatter to enter their password twice for verification. Below the text boxes will be a button labeled “Register".

## The register subsystem allows a chatter to create an account with the application that will allow them to communicate with the other chatters. Register will ask the chatter to enter in a name they wish to be known by on the application. This name will be the users’ id. Register will then ask them to enter a password two times. It will then verify that the passwords are the same. Once register accepts the id and password, it will create a Chatter object. Register will then check the registry to verify that an account has not already been made using the entered id. If the id does not match an existing account, the new account will be added to the registry and the user will be automatically logged into the application. If the id does match an existing account, register will notify the chatter that the id already exists and prompt them to enter a new id and reenter their password.

## Log In



## Figure 4.3

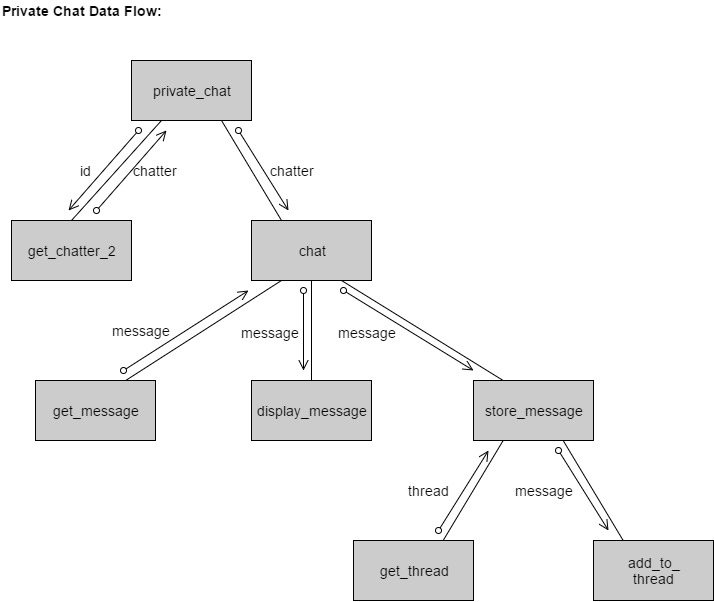
The interface for log in will be the main page. It will have a text box for chatters to enter their id’s and another text box below to enter their password. Underneath the two text boxes will be two buttons side by side. The button on the left will allow existing accounts to log in once they enter their information. The button next to that will be labeled “Register” which will direct them to the register page outlined in the previous section.

The log in subsystem will allow existing chatters to access their accounts. The chatter will be prompted to enter their id and password. Once the chatter clicks the log in button, their information will be checked against the registry. If their information matches an account, their status will be changed to mark them as logged in. Log in will then notify other logged in chatters of the newly logged in chatter by using the id of the chatter who logged in.

## Chat

There are two types of chat environments that chatters will have access to: one private chat environment and one public chat environment.

### Private Chat

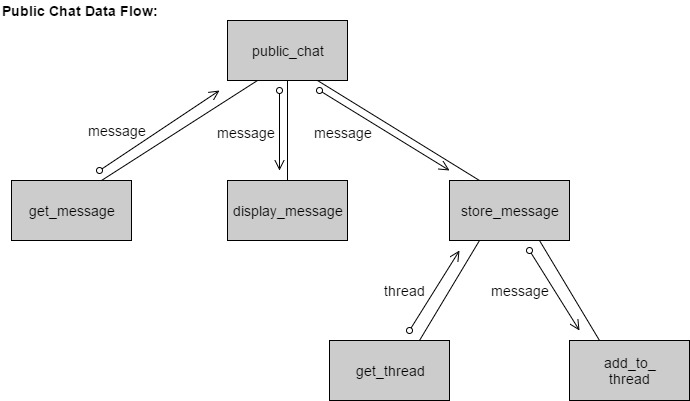


### Figure 4.4

The interface for private chat will consist of a section in the home page and a private chat page that allows two chatters to send and receive messages from one another. The home page will have a (scrollable section) which lists all chatters that are currently logged in. The private chat page will be the environment in which the two chatters will be able to communicate. It will consist of a display screen of previously sent messages that the chatter will be able to scroll up to see previous messages. On the bottom of the page will be a text bar for chatters to write their messages in. To the left of the text bar will be a send button.

The private chat subsystem allows a chatter to start messaging with another chatter. Chatter 1 will scroll through the list of chatters currently logged in. They will then click on the id of the other chatter they wish to send a message to. Private chat will then open a chat environment with both chatters included. Chatter 1 will type a message into the text box and hit the send button. The message will then be displayed in the message screen. Next, the message will be sent the message thread to be stored there for later access. Storing messages will be implemented using a binary search.

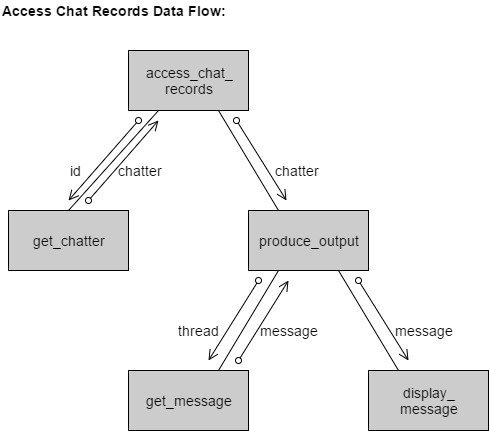
### Public Chat

**Figure 4.5**

The public chat interface will look identical to the private chat interface.

The public chat subsystem will add Chatter 1 to the interface. Chatter 1 will type a message into the text box and hit send. The message will then be displayed and stored in the public chat thread.

## Access Chat Records

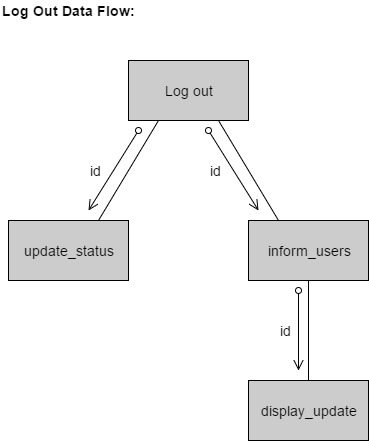


**Figure 4.6**

The access chat records subsystem interface will consist of a scrollable box. Inside the box will be links to threads from previous private chats. This will open another page that has a display screen for messages to be displayed.

The access chat records subsystem will allow users to view all their private chats with other chatters. By clicking on the interface link, access chat records will search chatter 1’s private threads and find the chatter who matches the id requested (chatter 2). It will then get each message in the thread and print it onto the display screen until all messages have been displayed.

## Log Out



**Figure 4.7**

The Log out subsystem will not have an interface. There will be a button in the home page for chatters to press when they wish to log off.

The log out subsystem will update the chatters status, changing them to logged off. It will then inform each chatter that is logged in that Chatter 1 has logged off.

# Detailed System Design

## Register

Detailed design of register function:

|  |  |
| --- | --- |
| Module name:  Module type:  Return type:  Input arguments:  Output arguments:  Error messages:  Files accessed:  Files changed:  Modules called:  Narrative: | **register**  Function  **void**  None  None  “error: id already exists.”  None  None  **get\_input**  arguments: None  **check\_registry**  arguments: **a\_chatter: Chatter**  **add\_to\_registry**  arguments: **a\_chatter: Chatter**  This module is invoked by the user to create an account. It will call **get\_input** which will get the id and two instances of the password, and verify the passwords match. **get\_input** will return a chatter object. **register** will then call **check\_registry** to verify that the new id is available. If id is available, **add\_to\_registry** will be called to add the chatter object to the database. |

Pseudocode for register function:

**void** register()  
{

Chatter a\_chatter *is* new Chatter(get\_input());

boolean valid *is* **false**;

**while**(valid *is* **false**)  
 {

**if**(check\_registry(a\_chatter) is **null**)

*use designated chatter subcontroller to call*add\_to\_registry(a\_chatter);

**else**{

*print* “error: id already exists.”

a\_chatter *equal* get\_input();

}

}

}

**Chatter** get\_input()

{

String id;

String p\_word1;

String p\_word2;

boolean valid *is* **false**;

**while**(valid *is* **false**)

{

id *gets* read\_id();

p\_word1 *gets* read\_passwords();

p\_word2 *gets* read\_passwords();

valid *gets* verify\_passwords(p\_word1, p\_word2);

}

**return** new Chatter(id, p\_word1);

}

## Log In

Detailed design for log\_in function:

|  |  |
| --- | --- |
| Module name:  Module type:  Return type:  Input arguments:  Output arguments:  Error messages:  Files accessed:  Files changed:  Modules called:  Narrative: | **log\_in**  Function  **void**  None  None  “error: Invalid id or password.”  None  None  **get\_input**  arguments: None  **check\_registry**  arguments: **a\_chatter: Chatter**  **inform\_users**  arguments: **id: String**  This module is invoked by the user to log in to the application. It will call **get\_input** which will get the users id and password. **get\_input** will return a chatter object. **log\_in** will then call check registry to verify that the account already exists. Next, **inform\_users** will be called which will alert currently logged in users that a chatter has logged in. |

Pseudocode for log\_in function:

**void** log\_in()

{

Chatter a\_chatter *is* new Chatter(get\_input());

boolean valid *is* **false**;

**while**(valid *is* **false**)

{

**if**(check\_registry(a\_chatter) *is* **true**)

{

inform\_users(a\_chatter.id);

valid *is* **true**;

}

**else**

*print* “error: Invalid id or password.”

}

}

//different from register get\_input

**Chatter** get\_input()

{

String id;

String p\_word;

id *gets* read\_id();

p\_word *gets* read\_password();

**return** new Chatter(id, p\_word);

}

## Chat

Chat is broken up into two subsystems: private chat and public chat.

### Private Chat

Detailed design for private\_chat function:

|  |  |
| --- | --- |
| Module name:  Module type:  Return type:  Input arguments:  Output arguments:  Error messages:  Files accessed:  Files changed:  Modules called:  Narrative: | **private\_chat**  Function  **void**  **id: String**  None  None  None  None  **get\_chatter**  arguments: **id: String**  **chat**  arguments: **a\_chatter: Chatter**  This module takes the id of chatter 2 that chatter 1 wishes to message privately and calls **get\_chatter** to get chatter 2’s object. It then calls **chat** to start a private chat between chatter 1 and chatter 2. |

Pseudocode for private\_chat function:

**void** private\_chat(String id)

{

Chatter a\_chatter *is* get\_chatter(id);

chat(a\_chatter);

}

### Public Chat

Detailed design for public\_chat function:

|  |  |
| --- | --- |
| Module name:  Module type:  Return type:  Input arguments:  Output arguments:  Error messages:  Files accessed:  Files changed:  Modules called:  Narrative: | **public\_chat**  Function  **void**  None  None  None  None  None  **get\_message**  arguments: None  **display\_message**  arguments: **message: String**  **store\_message**  arguments: **message: String**  This module calls **get\_message** and returns the message sent. Next, it calls **display\_message** to print the message on the chat display screen. Then it calls **store\_message** to store the message in the public chat thread. |

Pseudocode for public\_chat function:

**void** public\_chat()

{

String message;

message *gets* get\_message();

display\_message(message);

store\_message(message);

}

## Access Chat Records

Detailed design for access\_chat\_records function:

|  |  |
| --- | --- |
| Module name:  Module type:  Return type:  Input arguments:  Output arguments:  Error messages:  Files accessed:  Files changed:  Modules called:  Narrative: | **access\_chat­­\_records**  Function  **void**  **id: String**  None  None None  None  **get\_chatter**  arguments: **id: String**  **produce\_output**  arguments: **a\_chatter: Chatter**  This module calls **get\_chatter** to get the chatter object using the chatters id. It then calls **produce\_output** to print out all the messages from the chat thread. |

Pseudocode for access\_chat\_records function:

**void** access\_chat\_records(String id)

{

Chatter a\_chatter *is* get\_chatter(id);

produce\_output(a\_chatter);

}

## Log Out

Detailed design for log\_out function:

|  |  |
| --- | --- |
| Module name:  Module type:  Return type:  Input arguments:  Output arguments:  Error messages:  Files accessed:  Files changed:  Modules called:  Narrative: | **log\_out**  Function  **void**  None  **id: String** None  None None  **update\_status**  arguments: **id: String**  **inform\_users**  arguments: **id: String**  This module calls **update\_status** to update the chatter object as being logged out. Next, it calls **inform\_users** to notify other logged in users that the chatter has logged out of the application. |

Pseudocode for log\_out function:

**void** log\_out(String id)

{

update\_status(id);

inform\_users(id);

}