Lounge Group Chat

Nick Stanford Lead Coder
Erik Fong Concept Designer
Khalid Adkins Debugging/Secretary
Sharvita Paithankar Gui Designer/Coordinator

Table of Contents

Project Description and Background

Our program is a group chat called “Lounge,” and just like the name implies it allows anyone in the United States to join our lounge and chat with other users. Chat room users are allowed to connect to the lounge app anonymously and talk to other users about topics that they may not feel comfortable talking about in their everyday lives. The added functionality of image uploading gives an additional element of entertainment to the user’s experience. Since there is only one room anyone can join and give their feedback on the current topic of the lounge.

System Design Description

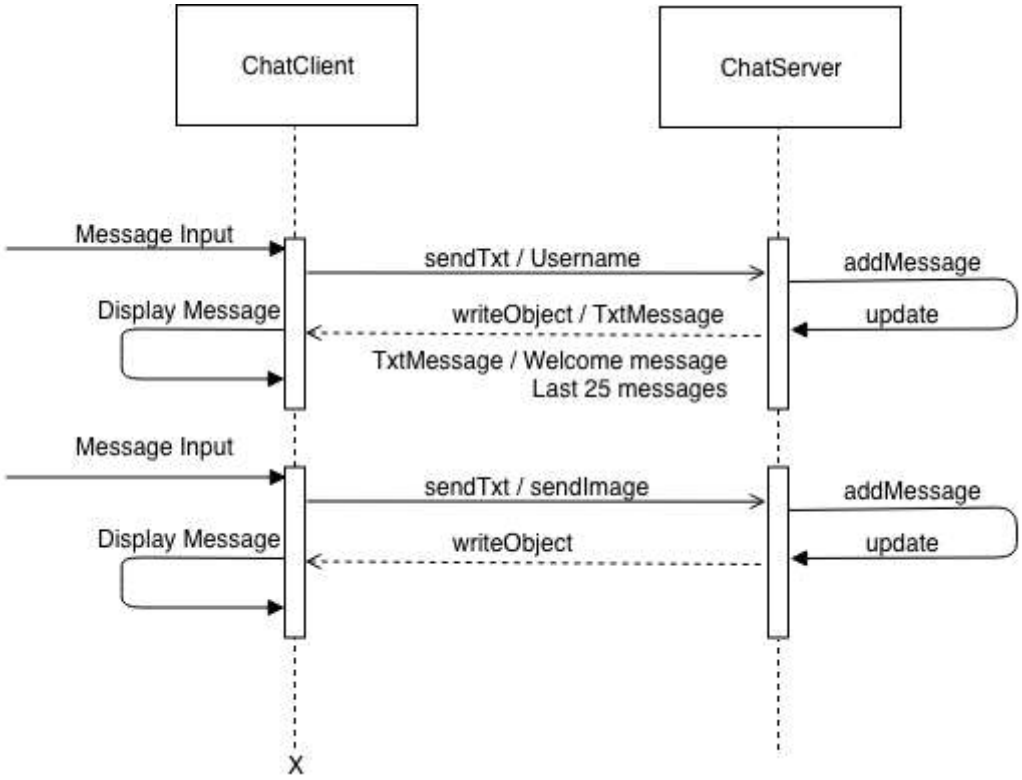
Our code was programmed in Java and we used the JavaFX to create the Graphical User interfaces. Currently our application is being run from from our local computer but we might upload our code to roku or one of the servers at Nick’s new employer, Indra, and have it on their servers. We leveraged Object Oriented programing principles create simple snippets of code that were integrated into the system to be able to be able to respond to the requests made by users.

The server side of our application uses the Factory and Observer patterns. The Factory pattern is used to create new threads for each client that accesses the server. These threads are created in the ChatUserHandler class and are assigned to MsgSender and MsgReceiver. The Observer pattern is used to update the shared ChatMessageLog for new users joining, users exiting, and for any message a user wants to send publically.

The client side and the Server side communicate from ChatClient to MsgReciever, and from MsgSender to MessageListener. The main class on the server side accepts incoming connections from new clients, then starts a new thread to handle that client’s socket and data streams, which in turn creates a MsgReceiver stream after the username has been set. After username creation, any time a client sends a message it gets added to the deque of messages the server has stored,

and every MessageSender in every thread gets notified that a new message should be sent. The client's MessageListener simply waits for incoming messages, decides what type of ChatMessage it's received, and outputs the object accordingly.

Sequence Diagram of Server/Client Relationship



At a fundamental level, the concept is simple, as shown by this sequence diagram. It becomes more complex when handling different types of ChatMessages.

Code Analysis

Examples of Multithreading/Multiprocessing/Concurrency on the Server

MessageDeleter.java: A process which checks the size of the ChatMessageLog twice per second. By deleting old messages, new users are not inundated with all the messages that have ever been sent.

```java
@Override
public void run() {
    while (!ChatServer.shutdown) {
        int currentSize = chatMessages.size();
        if (currentSize > 20) {
            while (currentSize > 20) {
                chatMessages.removeLast();
                currentSize--;
            }
        }
        try {
            Thread.sleep( millis: 500);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }
```

MsgReceiver.java: A runnable class. Every ChatUserHandler starts one of these threads after the user has successfully created a user name. It's member Boolean *shutdown* is a reference to the same one held by the listener and the ChatUserHandler, so when it receives a message to shutdown, it will modify all the shutdown Booleans on the threads corresponding to that user.

```java
@Override
public void run() {
    while(!shutdown && !ChatServer.shutdown) { // the first Boolean is user specific, the other would terminate all threads
        try {
            ChatMessage chatMessage = (ChatMessage) input.readObject();
            if (chatMessage instanceof TxtMessage && ((TxtMessage) chatMessage).equals("SHUTDOWN_NOW")) {
                chatMessage.setSender("");
                ((TxtMessage) chatMessage).setMessage(thisUser + " has left the room.");
                chatMessageLog.addMessage(chatMessage);
                ChatServer.userNames.remove(thisUser);
                this.shutdown = true;
            } else
                chatMessageLog.addMessage(chatMessage);
        } catch (SocketException se) {
            this.shutdown = true;
        } catch (Exception e) {
            e.printStackTrace();
        }
```

MsgSender.java: An Observer class that observes the ChatMessageLog and is updated when new messages are added to the queue wrapped by ChatMessageLog. Update is synchronized to make sure sending of messages to the user happens one at a time. Messages indicating that a user has joined or left are removed from the Log so that new users will not receive the arrival and

departure notifications that occurred prior to them logging on.

```
@Override
public synchronized void update(Observable o, Object arg) {
    if (!shutdown) {
        try {
            output.writeObject(chatMessageLog.getNewMessage());
        } catch (IOException e) {
            System.out.println("Exception thrown while sending message to " + thisUser);
        }
    }
    else {
        try {
            TxtMessage goodByeMsg = (TxtMessage)chatMessageLog.getNewMessage();
            output.writeObject(goodByeMsg);
            chatMessageLog.removeMessage(goodByeMsg);
        } catch (IOException e) {
            e.printStackTrace();
        }
        chatMessageLog.deleteObserver( o: this);
    }
}
```

ChatMessagLog.addMessage(): A synchronized function that ensures only 5 messages per second are added to the log. It was necessary to synchronize the block inside the function because synchronizing the function itself would lock down the entire instance of the class. That would prevent *MessageSender* from accessing member variable *newestMessage* when it observes that the ChatMessageLog has changed.

```
// threadsafe method only allows 5 incoming messages per second,
// but doesn't lock the entire instance so that Observers can use getNewMessage()
public void addMessage(ChatMessage aMessage) {
    synchronized (chatMessages) {
        this.newestMessage = aMessage;
        chatMessages.push(aMessage);
        setChanged();
        notifyObservers(aMessage.getSender());

        // only add 5 messages per second
        try {
            Thread.sleep( millis: 200);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }
}
```

In its construction, it is ChatMessageLog that starts the MessageDeleter Thread. Because MessageDeleter is such a simple routine, it was not necessary to pass the ExecutorService into ChatMessageLog. A simple one-off Thread works fine.

```
// initialize deque of messages, start the MessageDeleter
public ChatMessageLog() {
    this.chatMessages = new ConcurrentLinkedDeque<>();
    this.messageDeleter = new Thread(new MessageDeleter(chatMessages));
    messageDeleter.start();
}
```

Examples of Multithreading/Multiprocessing/Concurrency on Client Side

ChatClient.Java

This is one of two parallel threads that runs in the client class that is in charge of receiving messages on the client side. The first part allows the user to create a username. Messages are not sent to the server until the user confirms they are satisfied with their username. The second part (second screenshot, near the bottom) creates a listener for the client to begin continuously outputting messages from the server.

ChatClient pt1:

```
@Override
public void run() {
    ObjectInputStream input = null;
    try {
        input = new ObjectInputStream(serverConnection.getInputStream());
    } catch (IOException e) {
        e.printStackTrace();
    }
    // give the scene time to appear
    try {
        Thread.sleep( millis: 1000);
    } catch (InterruptedException e) {
        e.printStackTrace();
    }
    TxtMessage serverMsg = new TxtMessage( sender: "Server", message: " already exists ");
    printTxtMessage(new TxtMessage( sender: "Server", message: "Welcome user! Please enter a username."));
    while (serverMsg.contains("already exists")) {
        String confirm = "n";
        Controller.sendToServer = false;
        while (!confirm.equals("y\n")) {
            String lastLine = controller.getLastLine();
            while (controller.getLastLine().equals(lastLine)) { // wait for user to input text
                try {
                    Thread.sleep( millis: 100);
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
            }
            this.userName = controller.getLastLine();
```

Chat Client pt2

```
        printTxtMessage(new TxtMessage( sender: "Server",  message: "Are you satisfied with the username " +
                this.userName + "? (y or n)"));
        lastLine = controller.getLastLine();
        while (controller.getLastLine().equals(lastLine)) { // wait for user to input text
            try {
                Thread.sleep( millis: 100);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
        confirm = controller.getLastLine();
        if (!confirm.toLowerCase().equals("y\n")) {
            printTxtMessage(new TxtMessage( sender: "Server",  message: "Please enter a new username"));
        }
    }
    controller.setUsername(userName);
    Controller.sendToServer = true; // start having enter send txt to server
    try {
        output.writeObject(new TxtMessage( sender: "UnnamedUser", userName));
        serverMsg = (TxtMessage) input.readObject();
    } catch (Exception e) {
        e.printStackTrace();
    }
    printTxtMessage(serverMsg);
}
Thread listenThread = new Thread(new MessageListener(input, controller));
listenThread.start();
ChatMessage userMsg = new TxtMessage( sender: "",  message: this.userName + " has joined the chat room");
try {
```

Chat Client pt3:

The ChatClient waits until the controller is shutdown after the user exits the window, then tells the MessageListener to stop by changing its public static Boolean member *shutdown* to true.

```
    try {
        output.writeObject(userMsg);
    } catch (IOException e) {
        e.printStackTrace();
    }
    userMsg.setSender(this.userName);// Console only
    while (!ChatClient.shutdown) {
        if (Controller.shutdown) {
            try {
                output.writeObject(new TxtMessage(userName,  message: "SHUTDOWN_NOW"));
            } catch (IOException e) {
                e.printStackTrace();
            }
            ChatClient.shutdown = true;
        }
    }
    return;
}
```

MessageListener.Java: A Runnable class which reads ChatMessage objects from the server and prints them to the application window. The JVM throws exceptions when a non-fx thread accesses member functions of the controller. Therefore, the *printNewMessage()* function employs *Platform.runLater(Runnable arg),* a static method which adds a runnable method to the fx thread's queue of tasks. These tasks aren't guaranteed to run immediately, but they run soon enough for our application that there is no noticeable delay.

```
public MessageListener(ObjectInputStream objectInputStream, Controller controller) {
    this.objectInputStream = objectInputStream;
    this.controller = controller;
}

public void run() {
    ChatMessage chatMessage = null;
    while (!ChatClient.shutdown) {
        try {
            chatMessage = (ChatMessage)objectInputStream.readObject();
        } catch (Exception e) {
            e.printStackTrace();
        }
        printNewMessage(chatMessage);
    }
}

private void printNewMessage(ChatMessage chatMessage) {
    if (chatMessage instanceof TxtMessage) {
        TxtMessage txt = (TxtMessage)chatMessage;
        Platform.runLater(() -> controller.printTxt(txt)); // avoids exeptions from accessing controller
    }
    else if (chatMessage instanceof  ImageMessage) {
        ImageMessage img = (ImageMessage)chatMessage;
        Platform.runLater(() -> controller.printImage(img));
        // test for other types of messages
    }
}
```

Controller.java: ActionImageButton uses a FileChooser object to allow the user to select an image file from their computer, another example of concurrent operations.

```
public void actionImageButton() {
    FileChooser fileChooser = new FileChooser();
    Stage fileOpenStage = new Stage();
    fileOpenStage.setTitle("Choose an image file (JPEG, PNG, etc)");
    File file = fileChooser.showOpenDialog(new Stage());
    ImageMessage newImageMsg = new ImageMessage(username, file);
    sendImage(newImageMsg);
}
```

JavaFX containers require the use of a *Node* class to display objects in some part of your application window. Therefore, in order to display a BufferedImage, we need to construct an *ImageView*, a subclass of *Node. ImageView*'s constructor requires an Image Object; luckily, Java has a built in function, *SwingFXUtils.toFXImage(),* which does this conversion. PrintTxt() and sendTxt() both get text from the *inputTextArea* defined in the fxml file, and so need @fxml.

PrintImage, however, receives the image to be output as an argument, and so can modify the window through the *imagePane* and *images* objects

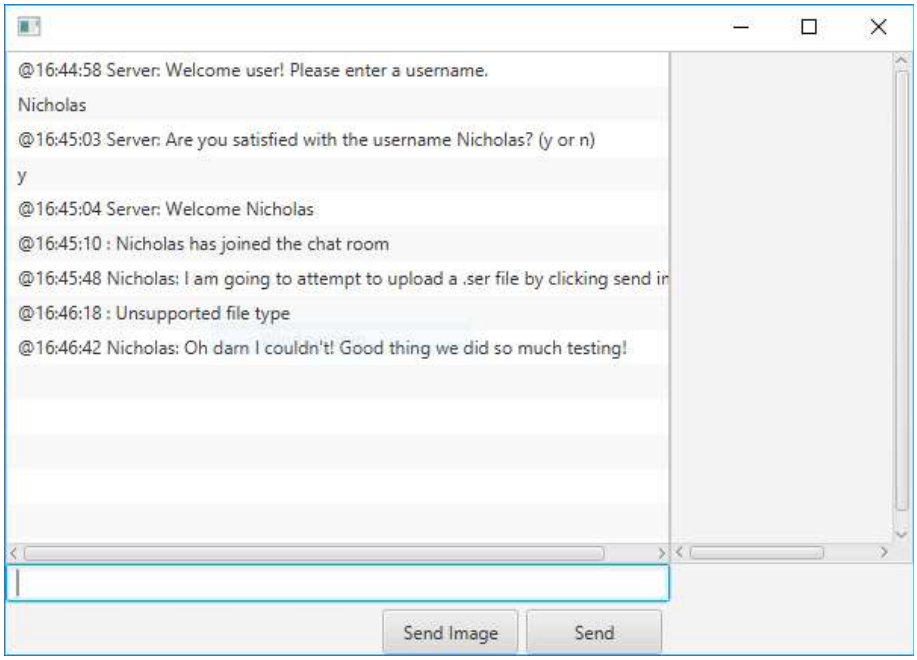
```
public void printImage(ImageMessage img) {
    ImageView imageView = new ImageView(SwingFXUtils.toFXImage(img.getImage(),  wimg: null));
    imageView.setPreserveRatio(true);
    imageView.setFitWidth(400);
    images.getChildren().add(imageView);
    imagePane.setContent(images);
}

@FXML
public void printTxt() {
    txtMessage = inputTextArea.getText();
    inputTextArea.setText(null);
    outputTextArea.getItems().add(txtMessage);
}

@FXML
public void sendTxt() {
    if (outputStream != null) {
        try {
            outputStream.writeObject(new TxtMessage(username, inputTextArea.getText().trim()));
        } catch (IOException e) {
            e.printStackTrace();
        }
        inputTextArea.setText(null);
    }
}
```

Tests

In the first stages we debugged our code using the system debugger. After we had a functioning application we debugged by having various users attempt to “break” the code. Closing the application improperly after it was not responding was one of the exceptions we had not anticipated and we had to write some code to handle that exception. If you stop the program by clicking the red square in the IDE, your username isn’t removed from the Server’s list of users, but normal users won’t have access to this if we figure out how to put our project in a compiled executable file. We didn’t use any bots, etc. to test our code, but due to robust exception handling we were able to identify problem areas.



Results

Our System is fully functional and can run multiple users at once. There is one bug where if the computer disconnects from the wireless internet and reconnects or the server is disconnected then the client receives an infinite loop of the last message they received.

Running the code is very simple and user friendly. Ensure that ChatServer is running and that port forwarding is enabled on port 7777 of your router. This should not be an issue at CU Denver, and may not be an issue because you will be using localhost on the client side. Now run the client. Enter a username and confirm that that is the name you want by entering "y". If you wish, run the client twice to simulate a two person chat room. Clients are able to send messages back and forth in the lounge by typing a message and pressing *enter* or clicking on the *send* button. Try sending a message by clicking *Send Image (.jpg).* The file explorer will appear, and you can choose any JPEG image. It will send immediately once a JPEG file has been opened.

Lessons Learned

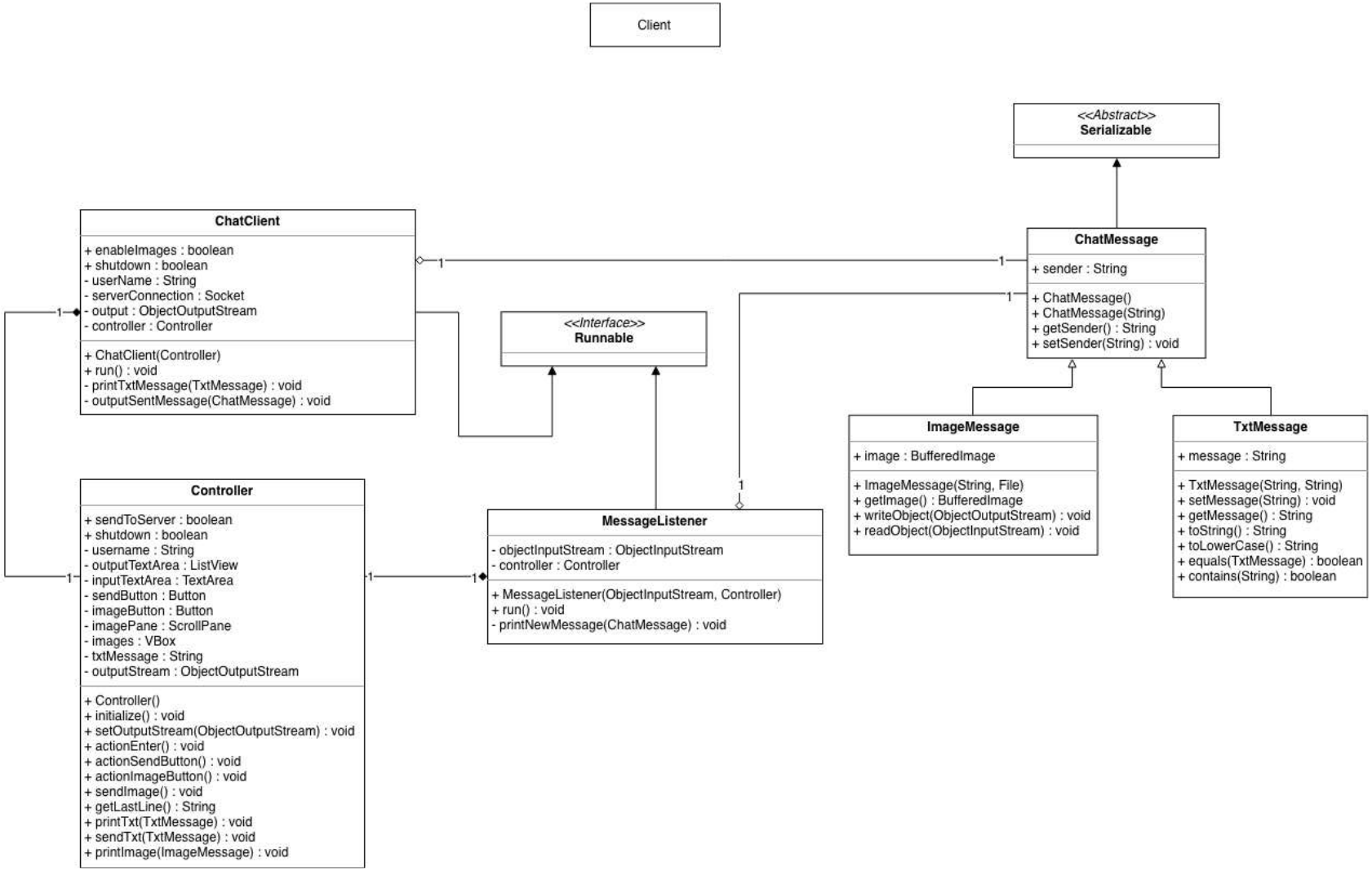
We learned that our team members are your most valuable assets. Dividing the large tasks into pieces for each person is the best way to tackle large problems. When making code, one should take care in making it understandable for your teammates, commenting wherever possible to

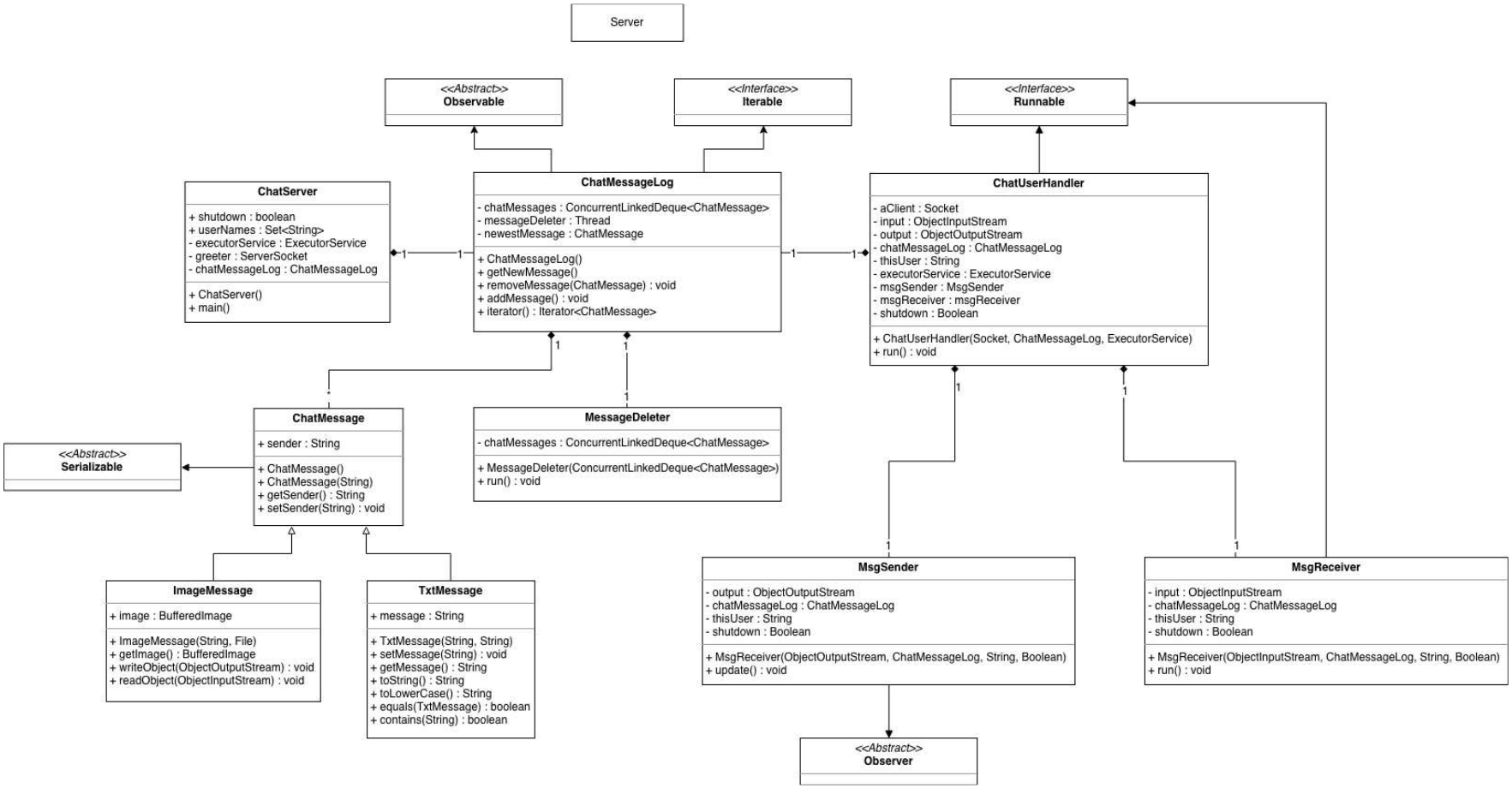
walk a reader through complex methods. The benefits of multithreading, and when it is applicable for a program to run well became clearer to us as we worked, and when it is and is not necessary to synchronize access to objects. We found that Java BufferedImages are not serializable, and taught ourselves how to override the writeObject() and readObject() methods by declaring the image objects transient so that they would be ignored by the default serialization. The concepts of .fxml files and controller access were new to us, and this was a great opportunity to learn some best practices in handling input from and output to a GUI. For example, the GUI must only be accessed by a JavaFX thread, and this was unknown to us until our program started throwing exceptions because we had passed our controller object into the classes underlying our Client Application. In the future, it would be best to use public static Booleans, observer/observable interactions, or piped input/output streams to tell the controller thread to access its own methods instead of calling those functions from other Threads. Short of that, our fix was to use the Platform.runLater() function to put inline-defined functions on the controller thread's action queue. This fix worked fine for our application, but the lack of immediacy from calling functions using runLater() caused some interesting side effects, requiring us to make the threads that used runLater() to then sleep for a fraction of a second to ensure that the controller could catch up. We learned that using screen builder is not as hard as it seems. Each button or a view can be connected to a specific variable or a method in the code. The method or code can be used to, for example, pass the message or picture from one text box to the other. We also removed our main class and controller from the default *sample* package that the IntelliJ IDE places them in to make sure that all the classes could see each other in the default package. In the future it may be more sophisticated to learn to properly implement package structures that allow classes access to the other classes they need, but not to those they don't. We think that employers will be impressed that we created an app that, with the addition of additional functionalities and a little polishing of the GUI, could pass for a production-level chat application.

Team Contributions

For this project, Nick was mainly involved in the coding of the server, and implementing the communication between the server and the client. Sharvita and Khalid were involved in

designing the GUIs and building the communication between the controller and the GUI. Erik has a talent for reading the sometimes illegible code written by others, and so simplified and segmented the code to make it readable.. Erik was the main person in designing the concept of the project and determining if it met qualifications of the rubric. Sharvita focused on making sure that the fxml file linked up with the controller, as well as educating herself on ActionEvents and making sure that the users enjoy a seamless experience when using the app. Khalid worked on debugging code and documentation of the project. Because we each worked on different parts of the code, Nick tied the various segments of code together so that they could function together.


Client
<<Abstract>>
Serializable
ChatClient
+ enableImages : boolean
+ shutdown : boolean
- userName : String
- serverConnection : Socket
- output : ObjectOutputStream
- controller : Controller
+ ChatClient(Controller)
+ run() : void
- printTxtMessage(TxtMessage) : void
- outputSentMessage(ChatMessage) : void
ChatMessage
+ sender : String
+ ChatMessage()
+ ChatMessage(String)
+ getSender() : String
+ setSender(String) : void
<<Interface>>
Runnable
ImageMessage
+ image : BufferedImage
+ ImageMessage(String, File)
+ getImage() : BufferedImage
+ writeObject(ObjectOutputStream) : void
+ readObject(ObjectInputStream) : void
TxtMessage
+ message : String
+ TxtMessage(String, String)
+ setMessage(String) : void
+ getMessage() : String
+ toString() : String
+ toLowerCase() : String
+ equals(TxtMessage) : boolean
+ contains(String) : boolean
Controller
+ sendToServer : boolean
+ shutdown : boolean
- username : String
- outputTextArea : ListView
- inputTextArea : TextArea
- sendButton : Button
- imageButton : Button
- imagePane : ScrollPane
- images : VBox
- txtMessage : String
- outputStream : ObjectOutputStream
+ Controller()
+ initialize() : void
+ setOutputStream(ObjectOutputStream) : void
+ actionEnter() : void
+ actionSendButton() : void
+ actionImageButton() : void
+ sendImage() : void
+ getLastLine() : String
+ printTxt(TxtMessage) : void
+ sendTxt(TxtMessage) : void
+ printImage(ImageMessage) : void
MessageListener
- objectInputStream : ObjectInputStream
- controller : Controller
+ MessageListener(ObjectInputStream, Controller)
+ run() : void
- printNewMessage(ChatMessage) : void
1


Server
<<Abstract>>
Observable
<<Interface>>
Iterable
<<Interface>>
Runnable
ChatServer
+ shutdown : boolean
+ userNames : Set<String>
- executorService : ExecutorService
- greeter : ServerSocket
- chatMessageLog : ChatMessageLog
+ ChatServer()
+ main()
ChatMessageLog
- chatMessages : ConcurrentLinkedDeque<ChatMessage>
- messageDeleter : Thread
- newestMessage : ChatMessage
+ ChatMessageLog()
+ getNewMessage()
+ removeMessage(ChatMessage) : void
+ addMessage() : void
+ iterator() : Iterator<ChatMessage>
ChatUserHandler
- aClient : Socket
- input : ObjectInputStream
- output : ObjectOutputStream
- chatMessageLog : ChatMessageLog
- thisUser : String
- executorService : ExecutorService
- msgSender : MsgSender
- msgReceiver : msgReceiver
- shutdown : Boolean
+ ChatUserHandler(Socket, ChatMessageLog, ExecutorService)
+ run() : void
<<Abstract>>
Serializable
ChatMessage
+ sender : String
+ ChatMessage()
+ ChatMessage(String)
+ getSender() : String
+ setSender(String) : void
MessageDeleter
- chatMessages : ConcurrentLinkedDeque<ChatMessage>
+ MessageDeleter(ConcurrentLinkedDeque<ChatMessage>)
+ run() : void
ImageMessage
+ image : BufferedImage
+ ImageMessage(String, File)
+ getImage() : BufferedImage
+ writeObject(ObjectOutputStream) : void
+ readObject(ObjectInputStream) : void
TxtMessage
+ message : String
+ TxtMessage(String, String)
+ setMessage(String) : void
+ getMessage() : String
+ toString() : String
+ toLowerCase() : String
+ equals(TxtMessage) : boolean
+ contains(String) : boolean
MsgSender
- output : ObjectOutputStream
- chatMessageLog : ChatMessageLog
- thisUser : String
- shutdown : Boolean
+ MsgReceiver(ObjectOutputStream, ChatMessageLog, String, Boolean)
+ update() : void
MsgReceiver
- input : ObjectInputStream
- chatMessageLog : ChatMessageLog
- thisUser : String
- shutdown : Boolean
+ MsgReceiver(ObjectInputStream, ChatMessageLog, String, Boolean)
+ run() : void
<<Abstract>>
Observer