COMP90015 2019 S2: Distributed Systems

Distributed Shared White Board

Group: Friday615

959698 Yixuan Tang; 971196 Fangwei Jiang; 986733 Pei-chun Kao;

986734 Chih-chi Chou; 1060766 Ruiyan Wang

1 Introduction

In this project, we are aimed at creating a shared whiteboard allowing one manager and multiple clients to draw and type text on it. The shared whiteboard allows users to chat with each other via a chat window and show active users. Also, the manager can see the information of all the clients and manage the access of clients, including allowing a specific client to join the whiteboard or kick a specific client out of the whiteboard. This shared whiteboard is achieved by implementing the Java RMI (remote message invocation) framework. Remote objects allow all the clients to access services provided by the centralized server as if manipulating the methods locally. As a result, all clients can share a real-time whiteboard, active user list and chat window without building connections with other clients.

2 Architecture

2.1 Server

In this project, a centralised server is utilised to provide RMI (remote message invocation) services and manage the status of all clients. The RMI services include update message, drawing panel and active user list to all clients. Also, the server is responsible to share information that needs to be broadcasted to other clients. That is to say, the RMI server acts as a message sharing centre to support the functionalities of a centralized shared whiteboard.

On the other hand, the server also provides an online user state table to trace whether a user is online or offline. The state of each user is dynamic recording because clients can be kicked by the manager or leave the whiteboard by themselves. Therefore, whenever a user left the whiteboard, the state table will update the behavior and add a new row to as “not active”. Since the user state table should be seen by the manager, it is reasonable to write a GUI function of state table on the server-side because when the server starts, the information of the manager will be subsequently released.

2.2 Client

The clients are designed to retrieve references of the RMI objects from the server. Also, methods defined in the client class allow the server to instantly update the drawing panel and share messages. Each client can broadcast panel and messages by executing the methods supporting by remote objects. This architecture simplifies communication processes and avoids concurrency issues between clients.

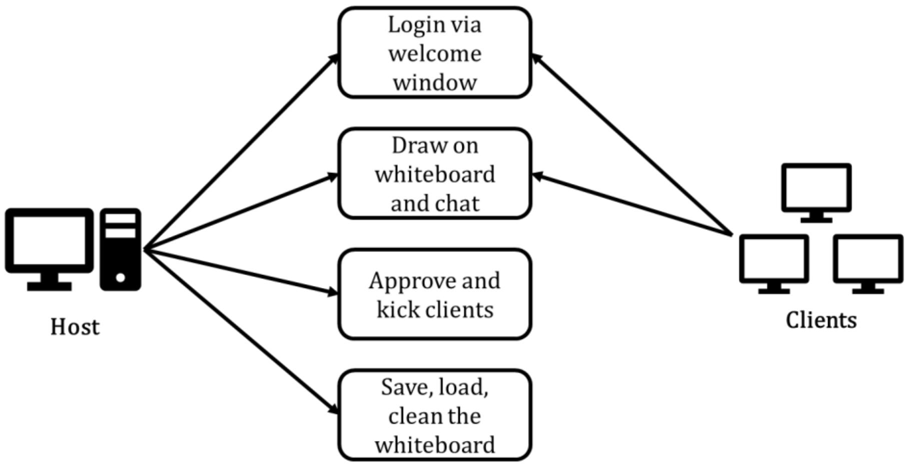


Figure 1. Server-Client architecture

2.3 Whiteboard Functionality

A shared whiteboard is designed to allow multiple users to draw concurrently. Similar to the existing drawing software, this shared whiteboard application can support functions such as pen, brush, a series of shapes, eraser as well as the text insertion. Four widths of line and a palette with pre-defined 16 colors enable users to easily exchange their ideas through drawing. Different font types and sizes are also implemented to make this application more practical.

2.4 Chat Functionality

The chat functionality is designed to achieve instant messaging (text only) for people who join the shared whiteboard. A user list is also built so everyone can be aware of who is sharing the whiteboard. When someone joins or leaves the whiteboard, a notification will be displayed in the chat window along with an update of the user list. The “kick-out” function is also carried out for the manager to expel users.

3 Communication Protocol

Remote message invocation (RMI) is utilised in this project to provide communication channels between clients and the server. The server has a client list that records all the active users. So, whenever a client performs an action such as draw new graphs or sends a message, it will execute the methods provided by the remote object to broadcast the draw panel or messages to all clients. This mechanism utilised proxy as a stub at the client end and provides transparency for clients. The main reason we choose RMI in this project is that comparing to TCP sockets, RMI handles the threads, sockets, object marshaling and unmarshalling for users. Therefore, messages can be exchanged between clients relatively easy compared to TCP sockets if the messages are serializable. Moreover, RMI makes communication between clients transparent because a remote object can be invoked as if it is a local object.

4 Implementation

4.1 Server RMI

There are two classes designed to achieve RMI server: Iserver and serverImp. Iserver is an interface that extends Remote and defines services provided to the clients, such as update client lists, drawing panel, server state table, and messages. Also, all the methods defined in Iserver throws RemoteException. On the other hand, ServerImp class extends UnicastRemoteObject and implements IServer. There are four major instance variables defined in this class: clientList, listeners, shapes, and index. The clientList is a variable saving the active users. Therefore, whenever the server broadcasts new graphs or messages, the server will check the client list and send the updating message one by one. The listeners is a variable that allows the server to instantly update the state table whenever a new user is joining or one old user is leaving the whiteboard. Moreover, serializable objects, shapes, and index are used to record the latest graphs drawn by the users. Since we utilize a shape array to store the graphs drawing on the whiteboard, an index is also necessary to notify clients which position should start with when drawing the next new graph. As a result, when a new graph is added to the whiteboard, shapes array and index are broadcasted by the server to update the shape array, index and drawing panel of each client.

4.2 Client RMI

There are two classes designed to achieve RMI client: Iclient and clinetImp. Iclient is an interface that extends Remote and defines methods to be called by the server to update each client's drawing panel, chat window and showing notifications when a user is kicked by the host or the host is disconnected. The clientImp class contains two instances variables: server and username. Since each client needs to broadcast the new graphs and messages via the server object, the server variable is necessary. Moreover, there are two types of methods defined in the clientImp: update and notification. Update methods such updatePanel, updateClient and updateMessage, are operated by the server the update the client ends. Notification methods are utilized to show messages when a user is kicked by the host or the host is disconnected.

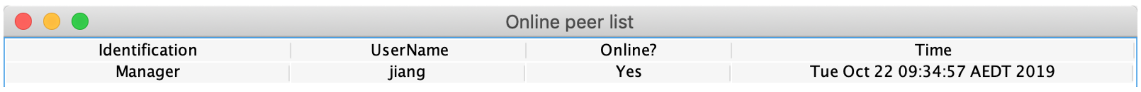
4.3 Basic Functionality

4.3.1 Welcome Window

Welcome window is designed as the portal of the whiteboard application. A user launches a whiteboard either as a manager/host or a client. The user who hosts a whiteboard will use his/her own IP address (localhost) as the server IP. Combined with a chosen port and a username, the host can register an RMI service and share this service with other users. On the other hand, a client can obtain the designated IP and port to connect a whiteboard service. There are error handlings to ensure the valid IP and port are offered. The username must be unique as well to identify each client.

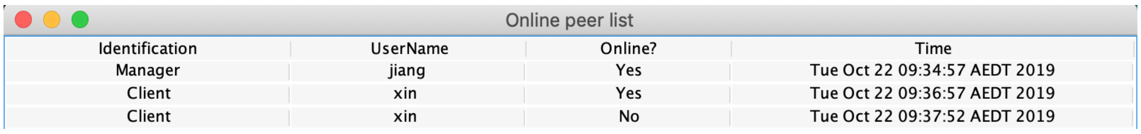
4.3.2 User State Table

Consider the simplicity of the code, the server should extend JFrame, which can directly call methods like setBounds and no need to use the same instance again and again. The most difficult part of designing the state table is how can we add or delete the row dynamically. In that scenario, AbstractTableModel would be a better solution. To use it, we should first import a package called "javax. swing. table. AbstractTableModel". After setting the basic layout using setDefaultCloseOperation, setBounds, getContentPane.setLayout, we can overwrite some methods in AbstractTablemodel, such as getValueAt, getColumnName, and getRowCount. Also, we import vectors and use them to capture each location. Once we have updated a row, fireTableStructrueChanged can be used to notify all listeners that the structure of the table has been altered. Moreover, for convenience, we decide to write a new function called resizeTable in order to change the size of the elements in the shape as the user stretches its margin. Since the time of conduct varies from each other, it would be better to record the time of each behavior in order to find the vulnerability by tracking the time. At this stage, we import java.util.Calendar and use a statement called Calendar to represents different times. The following two figures show the two conditions as the program is running (Figure 2 and Figure 3).



Identification	UserName	Online?	Time
Manager	jiang	Yes	Tue Oct 22 09:34:57 AEDT 2019

Figure 2. A manager named jiang has been appointed when the server starts.


Online peer list

Identification	UserName	Online?	Time
Manager	jiang	Yes	Tue Oct 22 09:34:57 AEDT 2019
Client	xin	Yes	Tue Oct 22 09:36:57 AEDT 2019
Client	xin	No	Tue Oct 22 09:37:52 AEDT 2019

Figure 3. Different states of one client named xin.

As shown in Figure 2, it only contains one row and shows the identification, username, online, time respectively. When the server starts, the list table will update the information of the manager by using the args[] in main method entered in the terminal by the user. Figure 3 indicates that when a new client named xin is added to the state table, the table will show whether the client is active or not. However, for rationality, a client should first login before quitting, which means he or she cannot directly quit without open a whiteboard.

4.3.3 User

User class defines the information that needs to be recorded for each client. The instance variables include ip, port, username, isHost, and client. Whenever a host accepts the join request made by a client, the client will execute a register client method provided by the remote object, and the server will update the user list saved on the server, and then broadcast the updated user list and a join message to every client.

4.4 Accessibility Management

The functionality of the manager plays an essential role in this project because it is associated with many aspects like user state table, different whiteboards using different threads and the concurrency between requests. The hardest part is how server can (manager) and clients (users) transfer message to handle the issue of different requests and responses. We should clearly know that this message transmission happens when the program is running, which means that there may be a wide variety of threads are active. In detail, the first transmission happens when a client run StartClient.java, ie., when the client presses the launch button to join an existing whiteboard, it sends a request to the server. The second transmission happens when a manager sends the responses back to the client. More precisely, it happens when the manager chooses yes or no on the jOptionpane to tell the program whether the client can join the whiteboard program. In that case, if we still using inputstream or outputstream, it will not work because the requests and responses can happen any time, but there is no timestamp of inputstream or outputstream.

In order to resolve this problem, we decide to add Eventlistener and EventObject and put them in chatServerListener and chatServerEvent separately. By doing this, we can now let the server class to implement chatServerListener, and call the addListener method defined in an interface called IServer, we also define notifyListener and notifyListenerQuit to handle different consequences when the client log in or out. These two methods can indirectly call serverEvent and serverEventQuit to change the structure of the user state table. As a result, when the first transmission happens, the client will call notifyListener and since server has already used addListener, it can directly know that a client wants to join the whiteboard. Secondly, when the second transmission happen, the server will pass an integer value n and send back to client, which shows whether the client can join or not. Moreover, when the server replies to the clients, it will only update the use state table as well as the userlist in the whiteboard to ensure the user information appeared in these two components are identical. If the data is not consistent, which will lead to no-pointer exception, even worse, it may crash on the program. Therefore, relevant exception and error handling procedures are implemented to ensure the two components are identical.

4.5 Whiteboard

4.5.1 Graph

All functions provided by the graph toolbar are completed by importing Graph 2D package and implementing the interface Serializable. We defined several different graph classes, such as pencil, circle, eraser, rectangle, etc. Each class extends the class Shape, which defines the variables of drawing properties including starting point, ending point, stoke, color, and text. Therefore, when a client clicks the functions listed in the whiteboard, it will trigger the chosen class and execute the draw method to generate a new graph on the drawing panel. The graph toolbar supports three different drawing functions: shapes, free draw and erase, and text inputting. Specific functions are listed as follows:

- Shapes: The whiteboard supports the users to draw lines. It also supports to draw shapes including ovals, circles, rectangles, round rectangles, and triangles or any of them filled with color.
- Free Draw and Erase: The whiteboard allows freehand drawing with a pencil or brush and freehand erasing.
- Text inputting: Users can type text anywhere in the drawing panel by clicking on the text button, entering text and selecting a location on drawing panel. Users can also choose the font size and font name from a given list.

4.5.2 Color Panel

There are two parts to the color panel. The upper one is a dual-bevel board with two buttons and below one is a palette with 16 pre-defined color buttons. The dual-bevel board allows a user to select two colors and display in the two bevel buttons. To select colors for the bevel buttons, a user should first click either one of them so the chosen one will be exhibited in the foreground. Then, indicate a color by pressing a color button in the palette. The dual-bevel board is a useful function to enable a user to draw with the two colors by pressing the left button and the right button on the mouse.

4.5.3 Whiteboard GUI

Whiteboard GUI integrates the functionalities mentioned above, including drawing panel, chat window, toolbar, text input, and file menu. This GUI enables users to work on a real-time whiteboard and timely sharing messages with others. Moreover, only the host has the authority to operate kick users, clean the drawing panel, save and load the whiteboard (*.wb) file. The whiteboard contains a file menu with function new, open, save, save as and close, which described as following:

- New: create a new drawing file. If the existing file is not saved, the whiteboard will ask the manager whether to save the file.
- Open: open an existing drawing file from the current computer.
- Save: save the file directly to the original file path. If the file is newly created and doesn't have a file path, the whiteboard will ask the manager to select the file path and file type.
- Save AS: save the file by selecting the file path and file type.
- Exit: close the whiteboard. If the existing file is not saved, the whiteboard will ask the manager whether to save the file.

When the host chooses "Save" or "Save As," he or she can decide to save the file in a jpg file or whiteboard file. While only the whiteboard files can be loaded for further editing. Whiteboard (*.wb) file is designed to save serializable shapes objects with the method "objectOutputStream," so when the host loads the ".wb" file, the host can share the shape array through RMI and update the drawing panel of all the clients.

4.6 Chat Functionality

There are three main elements in the chat functionality, the active user list, the chat window and the "kick-out" function.

4.6.1 Active User List

In this application, the active user is stored in an array list called "clientList". By retrieving the "clientList", the user list displays the number of active users and their usernames. When someone joins or leaves the whiteboard, the user list will be updated dynamically and then broadcasted to all users through the method defined in the server interface. Besides, a notification is presented in the chat window to inform the change of active users.

4.6.2 Chat Window

The chat window is implemented to achieve the instant message exchange for users who shared the whiteboard. A user can type the message in the texting box and trigger the transmission through the “Send” button or enter key on the keyboard. Again, the message will be broadcasted to all users through the method defined in the server interface. The chat window also helps to notify the change of users.

4.6.3 Kick-out Function

There are some functions can only be executed by the manager in this application, and one of them is the “kick-out” function. To be more specific, a manager can expel users by selecting a username in the user list and pressing the “Kick Out” button. This action will trigger the following outcomes:

- The user will be removed from “clientList” and an updated user list will be broadcasted to all users.
- A message will be broadcasted in the chat window to inform the action.
- A notification window will pop-up to inform the user that he/she has been kicked out from the whiteboard. The whiteboard is then forced to close.

5 Innovation

5.1 Innovative Features for Draw

- A “clear” method is applied to clean all the content on the whiteboard.
- A “brush” method is applied to draw with a freehand brush effect.
- The whiteboard allows the users to choose the line thickness to draw. There are four options provided in the drop-down menu.
- A series of options of font type and size are implemented to enrich the function of text insertion.
- The dual-bevel board allows users to draw with two different colors by clicking the left button and the right button on the mouse.
- The real-time location of cursor is displayed at the bottem.

5.2 Other Creation

- A “welcome window” is designed to involve an integrated portal for hosts and clients to launch the application.
- The active user list can display the number of online users.
- Chat window will show notification messages whenever a new user joins, leaves or

being kicked from the whiteboard.

6 Contribution

Student ID	Name	Contribution area	Contribution (%)
959698	Yixuan Tang	Design the file menu and help integrate different panels into the frame of Whiteboard GUI.	15
971196	Fangwei Jiang	Design the functionality of the manager as well as the dynamic user state table. Handle the concurrency issue between the above two. Assist in contributing code debugging of whiteboard and threads.	25
986733	Pei-chun Kao	Design the RMI server-client architecture and debug the concurrency issues of whiteboard and chat window. Assist teammates to design welcome window.	25
986734	Chih-chi Chou	Design the color panel, the chat window and welcome window. Help team to debug and optimize the GUI and functionalities comprehensively.	20
1060766	Ruiyan Wang	Design the toolbar for graph drawing and text input and menubar for file operations, thickness settings and instructions. Create images for buttons.	15

7 Conclusions

The purpose of this project is to design a shared whiteboard to enable multiple users to access concurrently. The whiteboard application is accomplished through RMI to achieve multi-thread. That is, the whiteboard service is registered by a host and a client can connect to the service via a remote object without building a connection to other clients. The functionalities of the whiteboard application contain drawing, text insertion, and message exchange. The host/manager can execute save, load and clear method to manipulate the draw panel. The host can also decide who can join the whiteboard and kick out users.

Appendix

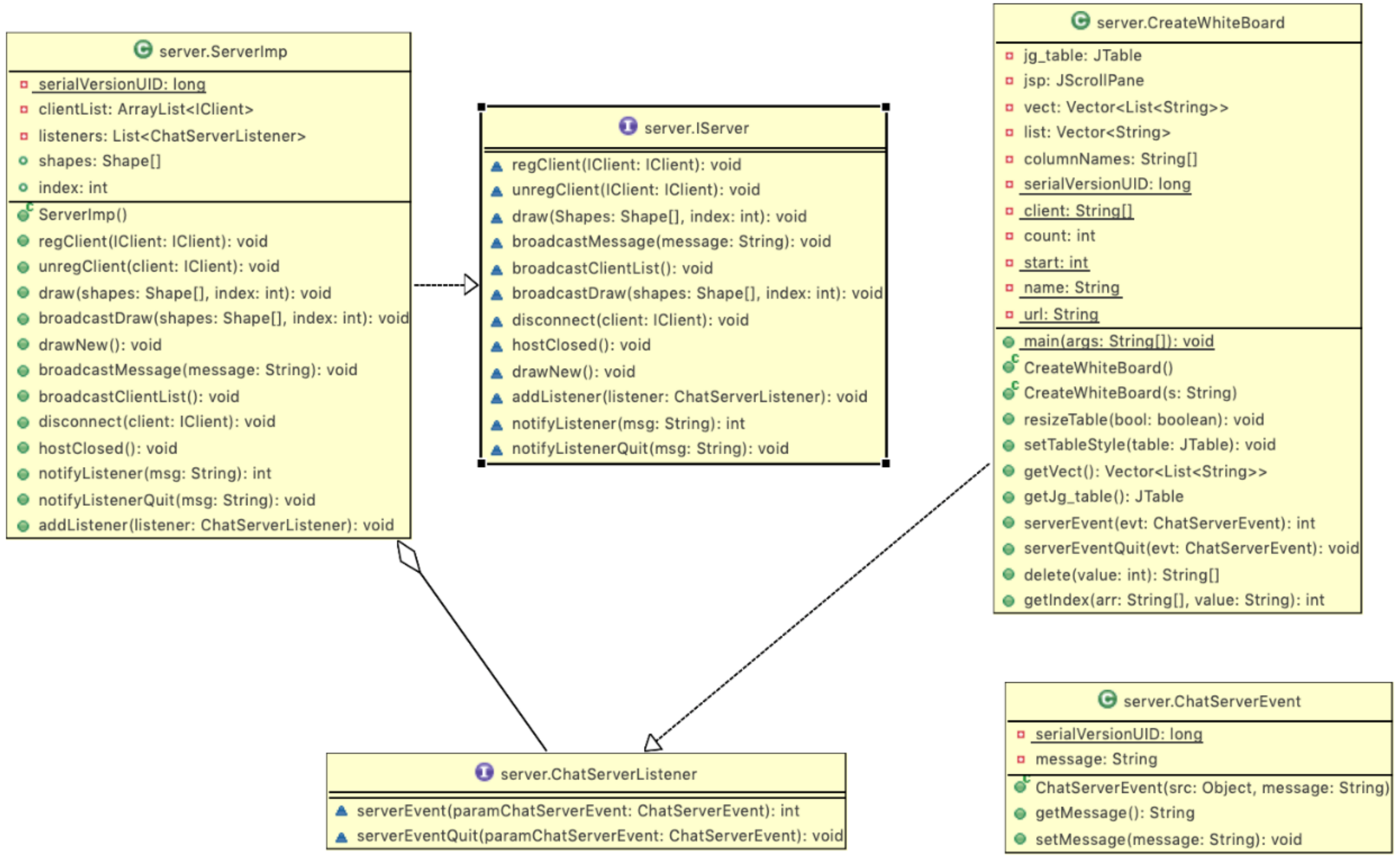

server.ServerImp
serialVersionUID: long
clientList: ArrayList<IClient>
listeners: List<ChatServerListener>
shapes: Shape[]
index: int
ServerImp()
regClient(IClient: IClient): void
unregClient(client: IClient): void
draw(shapes: Shape[], index: int): void
broadcastDraw(shapes: Shape[], index: int): void
drawNew(): void
broadcastMessage(message: String): void
broadcastClientList(): void
disconnect(client: IClient): void
hostClosed(): void
notifyListener(msg: String): int
notifyListenerQuit(msg: String): void
addListener(listener: ChatServerListener): void
server.IServer
regClient(IClient: IClient): void
unregClient(IClient: IClient): void
draw(Shapes: Shape[], index: int): void
broadcastMessage(message: String): void
broadcastClientList(): void
broadcastDraw(shapes: Shape[], index: int): void
disconnect(client: IClient): void
hostClosed(): void
drawNew(): void
addListener(listener: ChatServerListener): void
notifyListener(msg: String): int
notifyListenerQuit(msg: String): void
server.CreateWhiteBoard
jg_table: JTable
jsp: JScrollPane
vect: Vector<List<String>>
list: Vector<String>
columnNames: String[]
serialVersionUID: long
client: String[]
count: int
start: int
name: String
url: String
main(args: String[]): void
CreateWhiteBoard()
CreateWhiteBoard(s: String)
resizeTable(bool: boolean): void
setTableStyle(table: JTable): void
getVect(): Vector<List<String>>
getJg_table(): JTable
serverEvent(evt: ChatServerEvent): int
serverEventQuit(evt: ChatServerEvent): void
delete(value: int): String[]
getIndex(arr: String[], value: String): int
server.ChatServerListener
serverEvent(paramChatServerEvent: ChatServerEvent): int
serverEventQuit(paramChatServerEvent: ChatServerEvent): void
server.ChatServerEvent
serialVersionUID: long
message: String
ChatServerEvent(src: Object, message: String)
getMessage(): String
setMessage(message: String): void

Figure 1. UML of RMI server

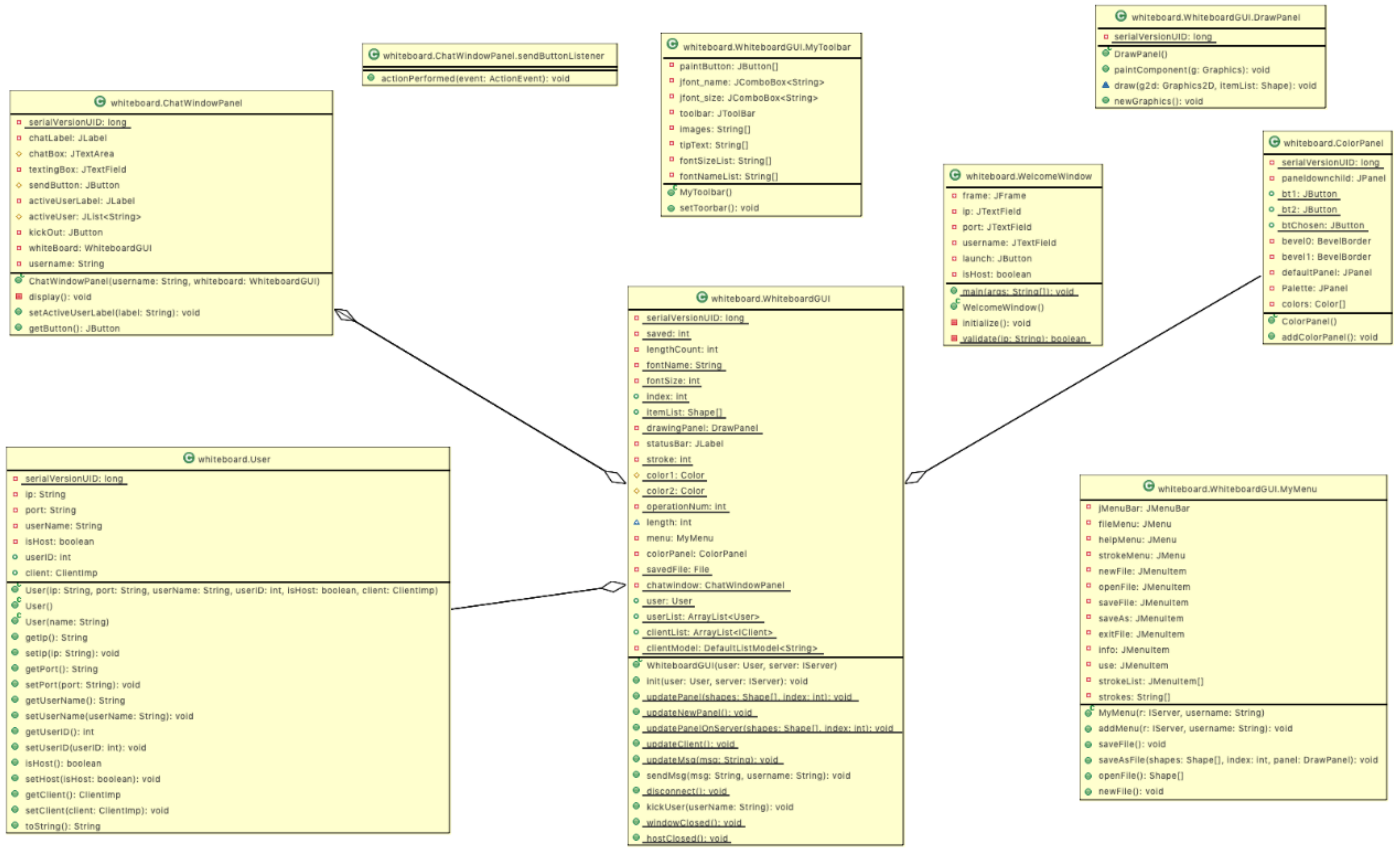


Figure 2. UML of Whiteboard GUI

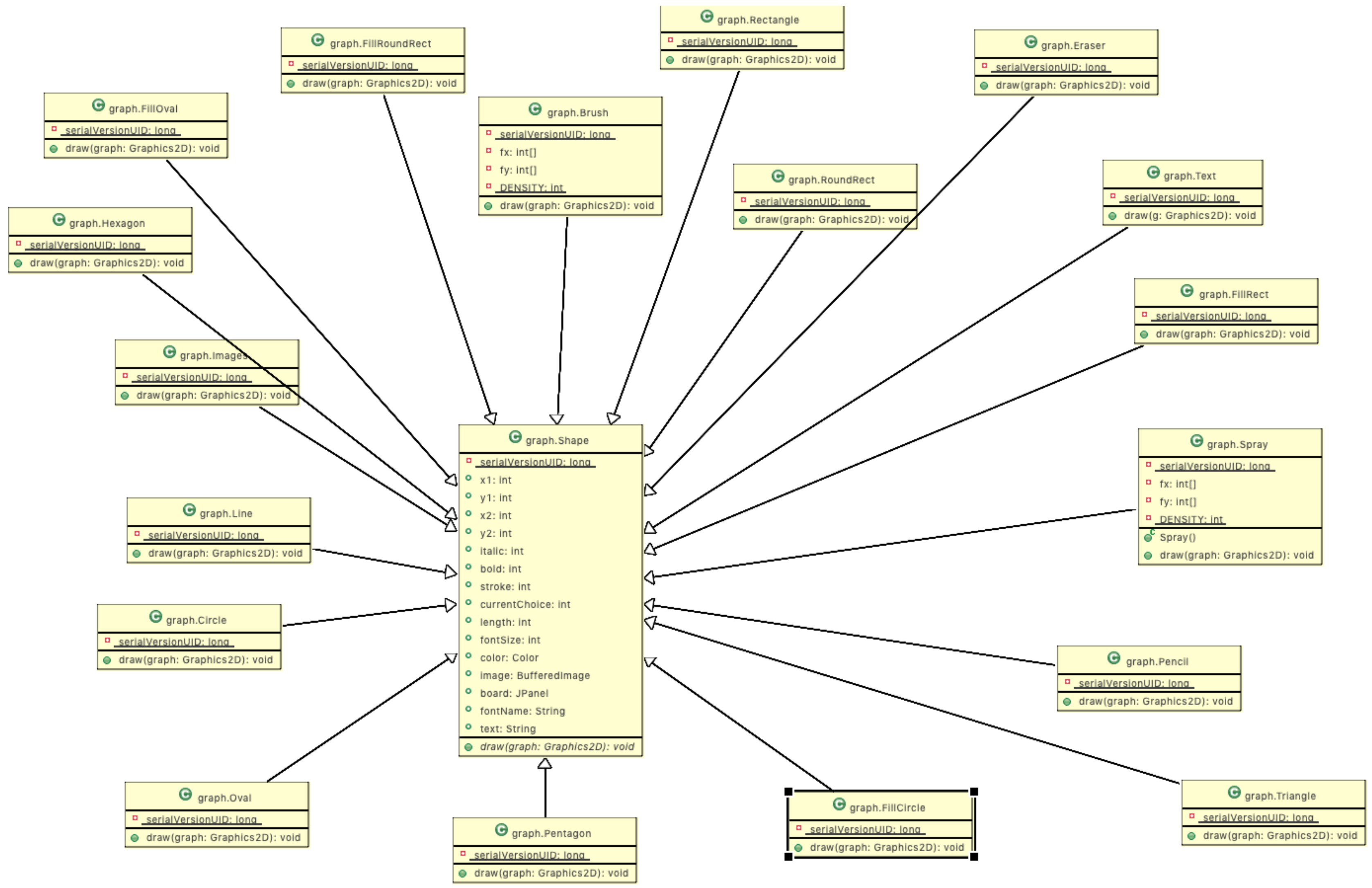


Figure 3. UML of graph

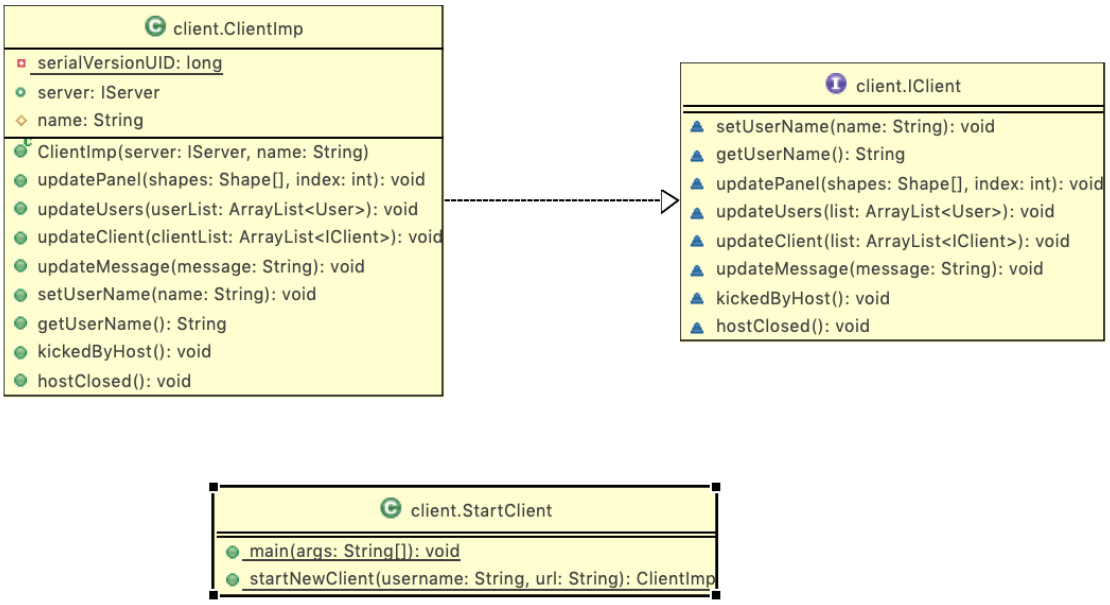
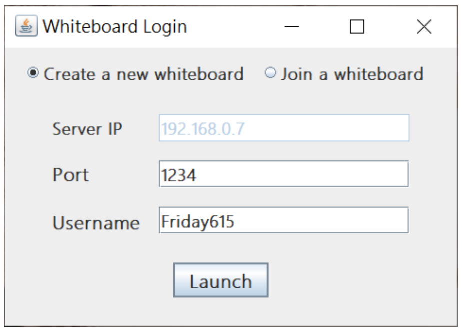

client.ClientImp
serialVersionUID: long
server: IServer
name: String
ClientImp(server: IServer, name: String)
updatePanel(shapes: Shape[], index: int): void
updateUsers(userList: ArrayList<User>): void
updateClient(clientList: ArrayList<IClient>): void
updateMessage(message: String): void
setUserName(name: String): void
getUserName(): String
kickedByHost(): void
hostClosed(): void
client.IClient
setUserName(name: String): void
getUserName(): String
updatePanel(shapes: Shape[], index: int): void
updateUsers(list: ArrayList<User>): void
updateClient(list: ArrayList<IClient>): void
updateMessage(message: String): void
kickedByHost(): void
hostClosed(): void
client.StartClient
main(args: String[]): void
startNewClient(username: String, url: String): ClientImp

Figure 4. UML of client


Whiteboard Login

[✓ Create a new whiteboard] Join a whiteboard

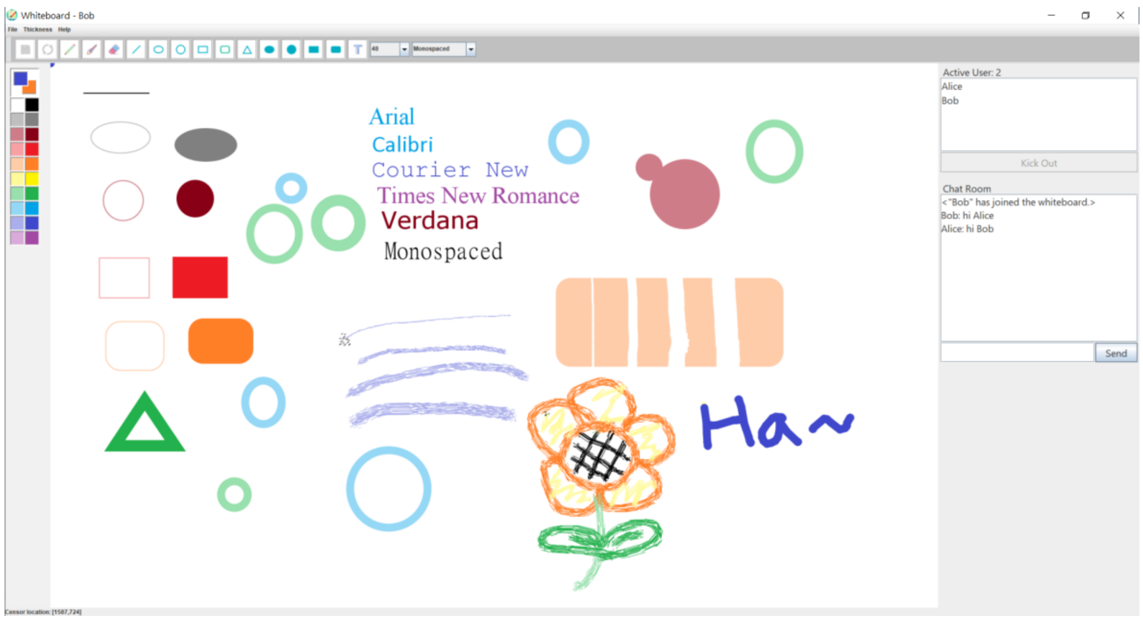
Server IP 192.168.0.7

Port 1234

Username Friday615

Launch

Figure 5. Welcome window



Active User: 2

Alice

Bob

Kick Out

Chat Room

<"Bob" has joined the whiteboard.>

Bob: hi Alice

Alice: hi Bob

Send

Figure 6. Whiteboard demonstration

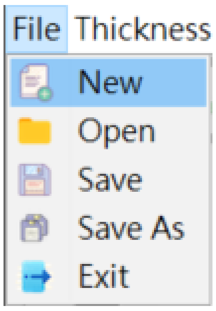


Figure 7. File menu

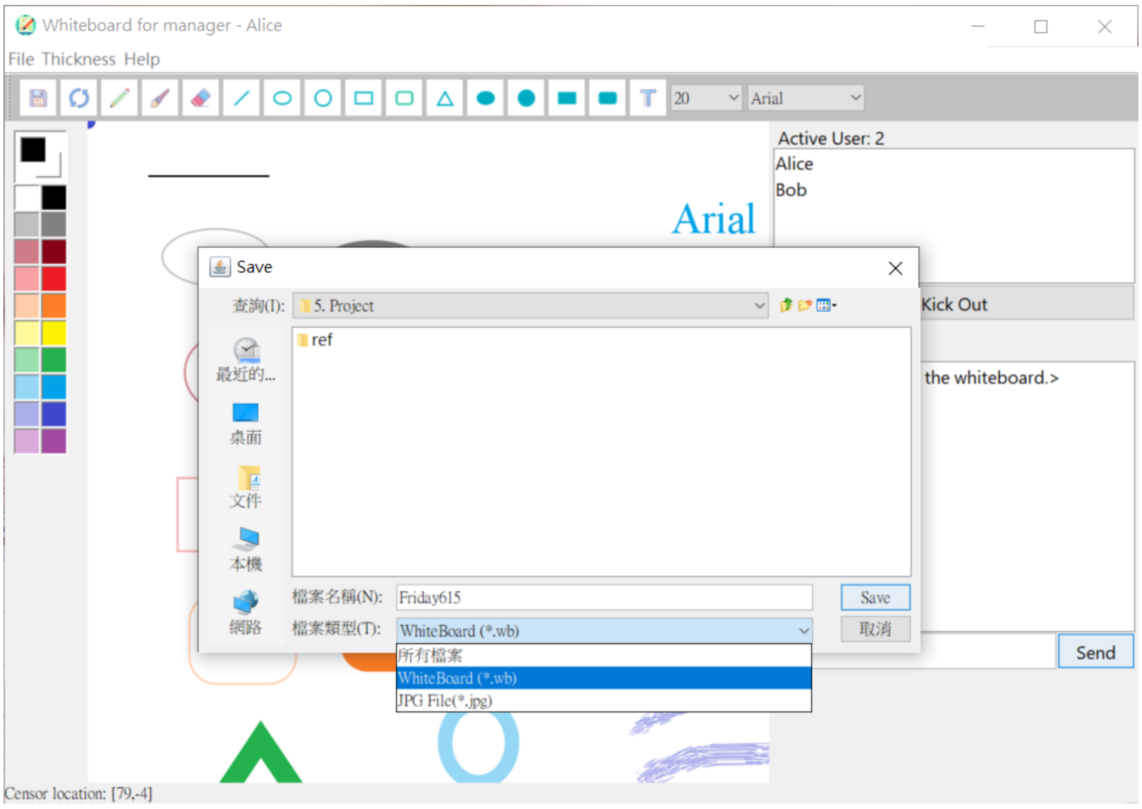


Figure 8. Dialog window of filechooser

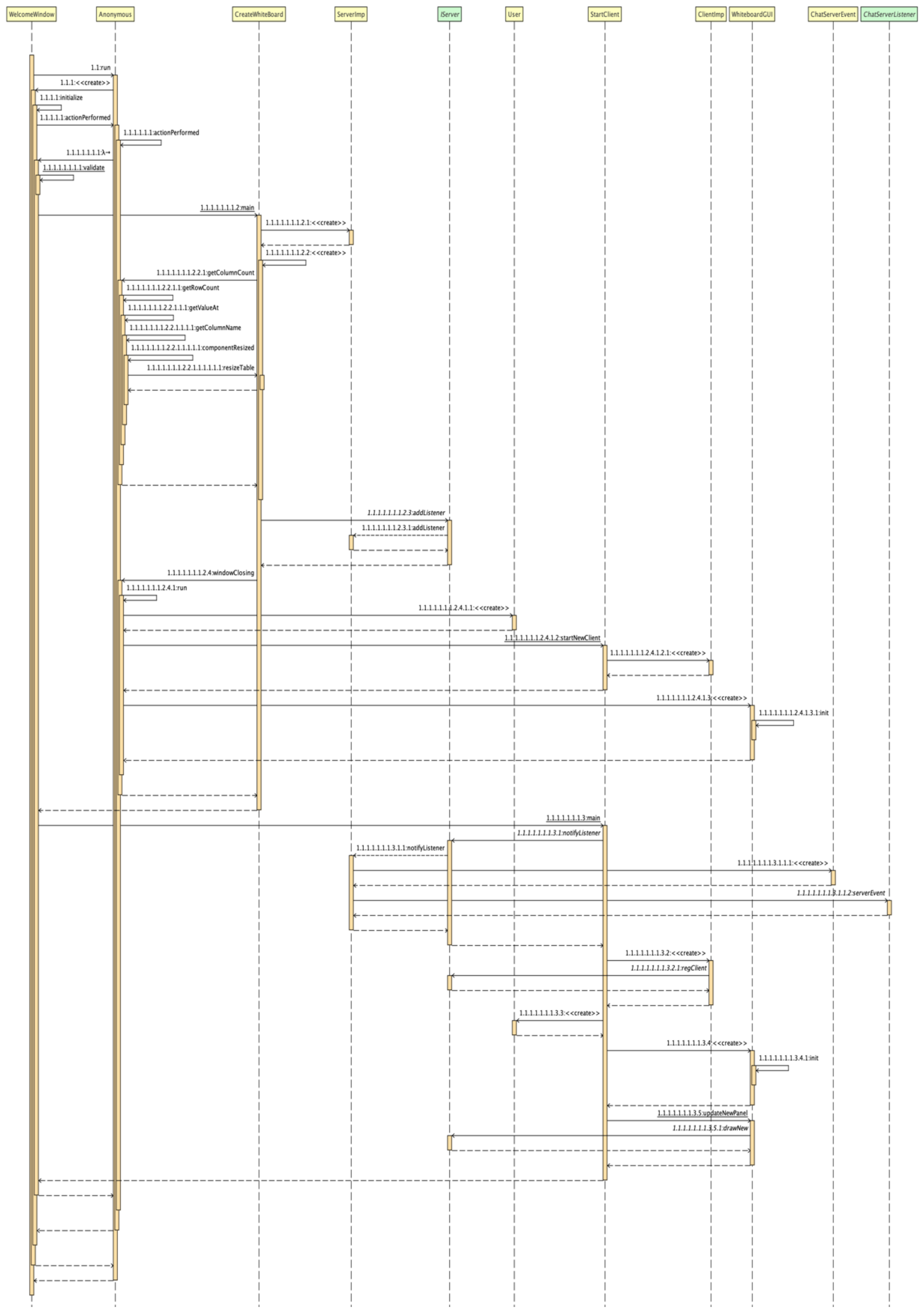

WelcomeWindow
Anonymous
CreateWhiteBoard
ServerImp
IServer
User
StartClient
ClientImp
WhiteboardGUI
ChatServerEvent
ChatServerListener
1.1:run
1.1.1:<<create>>
1.1.1.1:initialize
1.1.1.1.1:actionPerformed
1.1.1.1.1.1:actionPerformed
1.1.1.1.1.1.1:λ→
1.1.1.1.1.1.1.1:validate
1.1.1.1.1.1.1.2:main
1.1.1.1.1.1.1.2.1:<<create>>
1.1.1.1.1.1.1.2.2:<<create>>
1.1.1.1.1.1.1.2.2.1:getColumnCount
1.1.1.1.1.1.1.2.2.1.1:getRowCount
1.1.1.1.1.1.1.2.2.1.1.1:getValueAt
1.1.1.1.1.1.1.2.2.1.1.1.1:getColumnName
1.1.1.1.1.1.1.2.2.1.1.1.1.1:componentResized
1.1.1.1.1.1.1.2.2.1.1.1.1.1.1:resizeTable
1.1.1.1.1.1.1.2.3:addListener
1.1.1.1.1.1.1.2.3.1:addListener
1.1.1.1.1.1.1.2.4:windowClosing
1.1.1.1.1.1.1.2.4.1:run
1.1.1.1.1.1.1.2.4.1.1:<<create>>
1.1.1.1.1.1.1.2.4.1.2:startNewClient
1.1.1.1.1.1.1.2.4.1.2.1:<<create>>
1.1.1.1.1.1.1.2.4.1.3:<<create>>
1.1.1.1.1.1.1.2.4.1.3.1:init
1.1.1.1.1.1.1.3:main
1.1.1.1.1.1.1.3.1:notifyListener
1.1.1.1.1.1.1.3.1.1:notifyListener
1.1.1.1.1.1.1.3.1.1.1:<<create>>
1.1.1.1.1.1.1.3.1.1.2:serverEvent
1.1.1.1.1.1.1.3.2:<<create>>
1.1.1.1.1.1.1.3.2.1:regClient
1.1.1.1.1.1.1.3.3:<<create>>
1.1.1.1.1.1.1.3.4:<<create>>
1.1.1.1.1.1.1.3.4.1:init
1.1.1.1.1.1.1.3.5:updateNewPanel
1.1.1.1.1.1.1.3.5.1:drawNew

Figure 9. Sequence Diagram