COMP90015 Distributed Systems

Assignment 2 Sem 1 2021 Report

Distributed Whiteboard

Name **:** Chan Jie Ho

Student ID **:** 961948

Email **:** chanjieh@student.unimelb.edu.au

Word Count **:** 3011

TABLE OF CONTENTS

1. Project Scope

With large systems and applications, it would be unlikely that all the components of the system would be hosted on the same device. This is especially true these days with the advancement of cloud technology and most applications being hosted on the webs while requiring separate servers for their database, all of which may be running on different machines. This makes proper communication and error handling all the more important.

With cloud technology, it is also important to be able to allow for the integration and communication of different systems, or codes. For example, a client application, server system, and database system have vastly different behaviours and data that they need, thus requiring different method calls for different systems.

Socket programming could help facilitate this communication, however the complexity of the system grows with the size of the system, especially with different methods being called by different components. Error handling also becomes much more complicated in such applications. Remote Method Invocations (RMI) greatly aids us in this situation as they allow method invocations between objects in different processes on different machines like as if it was on their own machine. RMI also supports an at-most-once method invocation semantic that ensures methods are called and executed exactly once, providing the caller with the response, or not at all, which would lead to the caller receiving the exception.

In order to simulate such an environment, this project was tasked with designing and implementing a shared whiteboard that needs to allow multiple users to draw a range of shapes and texts simultaneously on a canvas with additional features such as a user account and management, and chat system. Proper communication between the systems is mandatory as the canvas would act as a shared resource that all the clients and servers would need to access and modify. RMI was used in this project to facilitate communication between the different components of the system. The system was created using Java while the client GUI was created using Java Swing Designer.

2. Supported Features

Below is an exhaustive list of features that the system should be able to support as outlined in the project specification:

2.1. Basic Features Supported

- Multiple users can draw on a single interactive canvas shared between all users
- Users can draw a line, circle, oval, and rectangle
- Users can input text into the whiteboard
- Users can choose from at least 16 colours available to draw with
- User account or username based system to uniquely identify users
- Users can see who is currently editing the same whiteboard
- Users can connect and disconnect to their discretion
- Real-time drawing, without delays between making and observing edits

2.2. Advanced Features Supported

- Text-based chat window to allow users to communicate with each other
- Manager-based approval system
- Managers have file methods to create, save, open, or close a whiteboard
- Managers can kick other users.

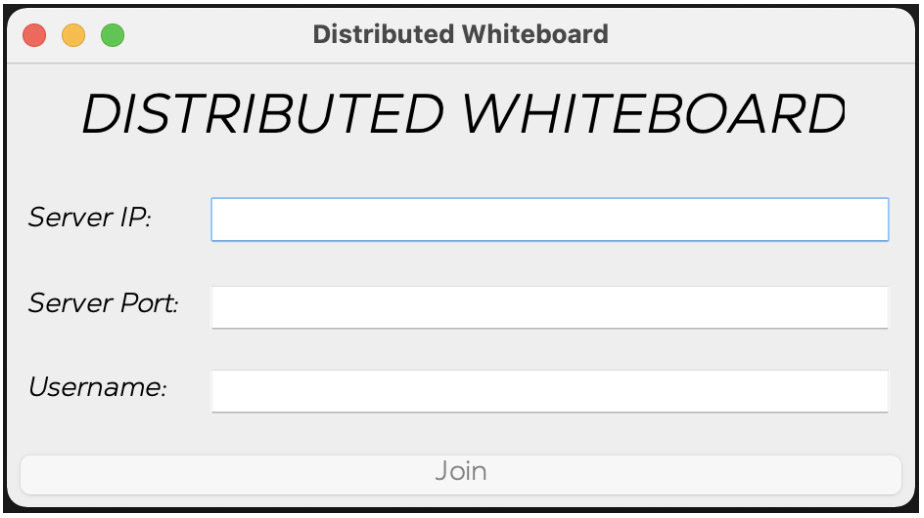
The implementation of these features will be discussed in ***Section 3***.

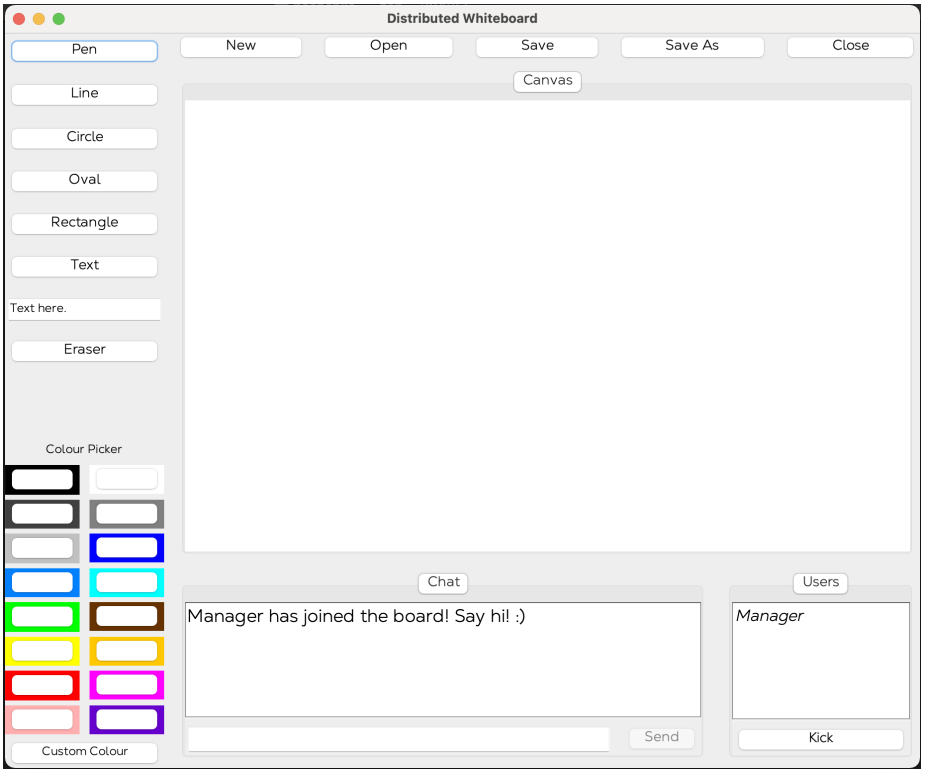
3. Basic Features

Below details the basic features implemented.

3.1. Client GUI

Upon launching of the client, a lobby is displayed prompting the user to input a username as well as the IPv4 address and port number of the whiteboard server and database that the user wishes to join. If provided valid inputs are provided and a connection is successfully established, the whiteboard GUI will then be displayed. The GUI will have all the canvas, drawing and colour tools, an active-user list, and other features (presented in ***Section 4***).



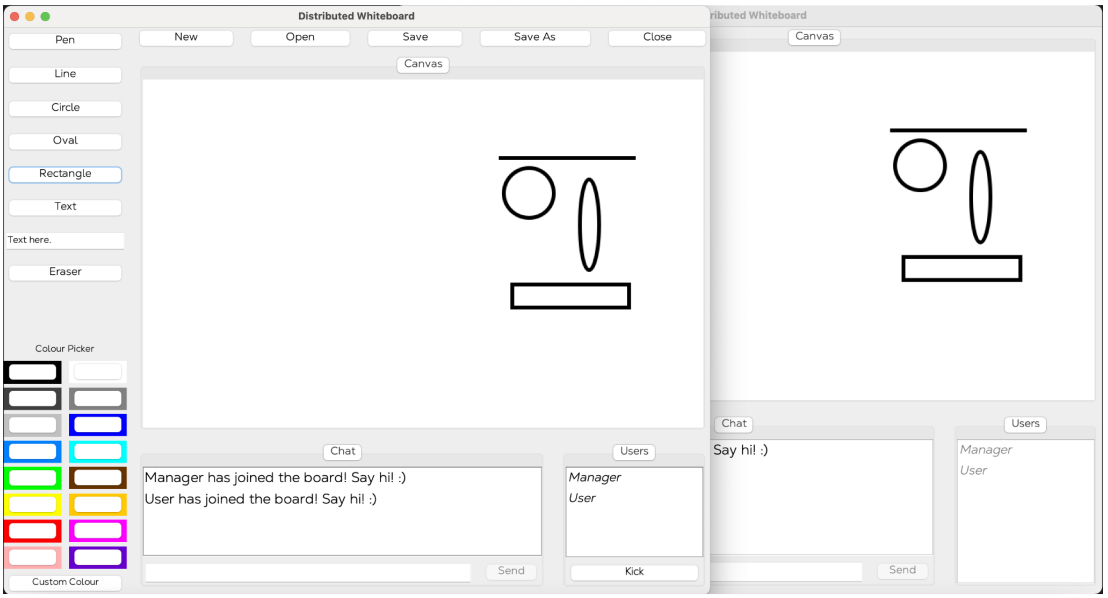


3.2. Concurrent Access and Real-Time Editing of Single Shared State

All users will have the same whiteboard state shown to them on their GUI. All drawings made by a user are immediately shown to all other users upon releasing of the mouse – with the exception of the two tools (see ***Section 7***).

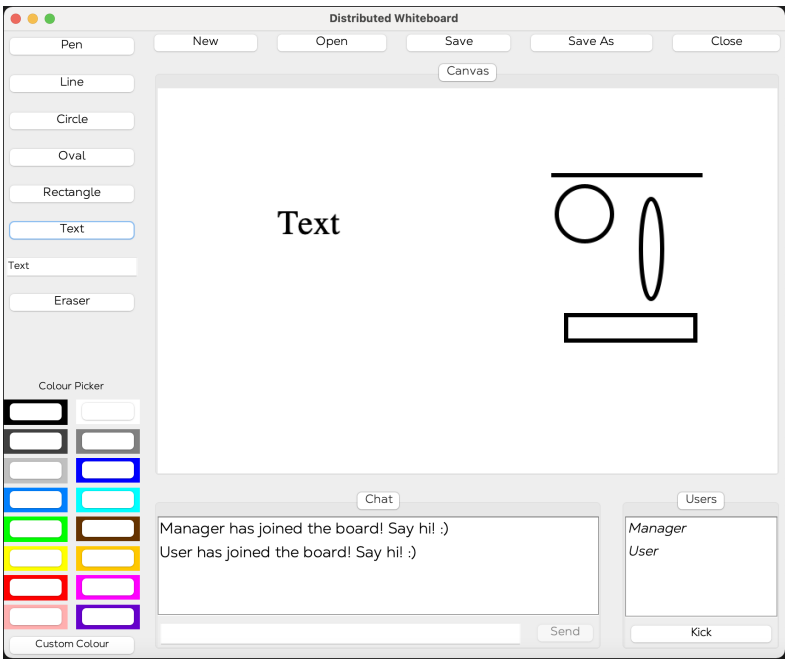
3.3. Basic Shapes

Users can create a line, circle, oval, and rectangle wherever they like on the board.



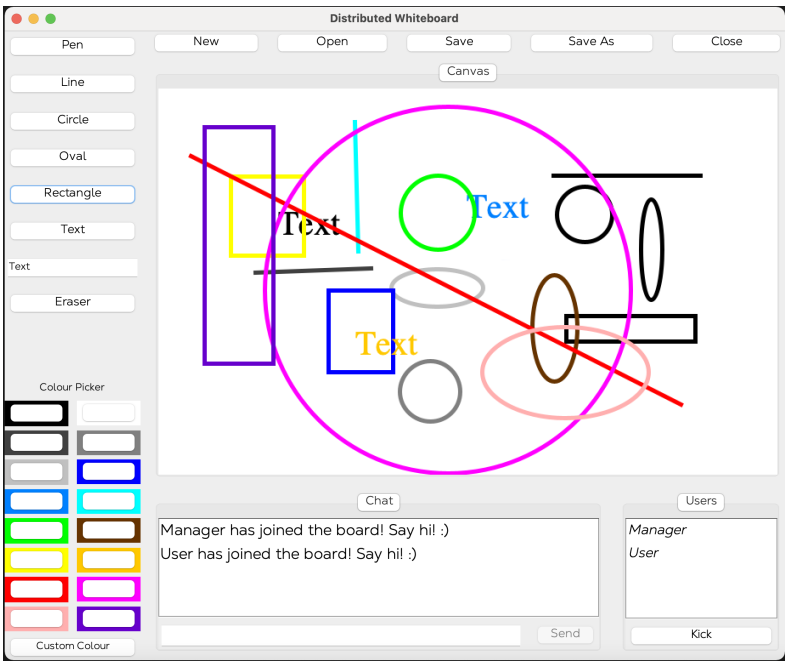
3.4. Text Input

If the Text tool is chosen, clicking on the canvas will display the text in the provided text box at that spot on the canvas.



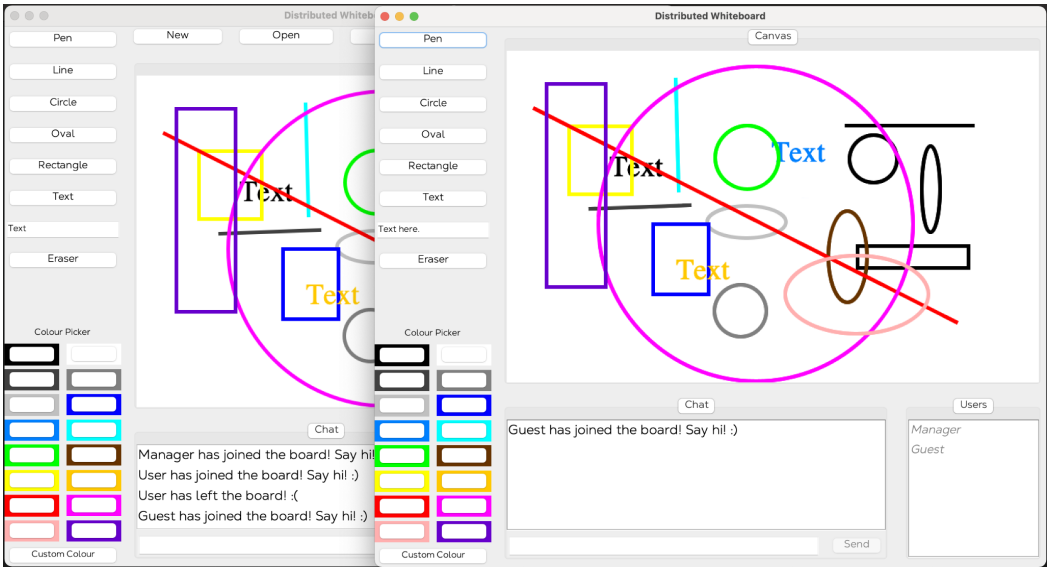
3.5. Colours

16 default colours are made available to the users to draw the shapes and text in whichever colour they choose.



3.6. Connection and Disconnection

Users are able to leave the whiteboard at any point in time, and when joining again in the future, the GUI will show the current version of the whiteboard. As shown below, despite Guest only joining after the board had been drawn on, the drawings were still shown on Guest's whiteboard.



3.7. Username

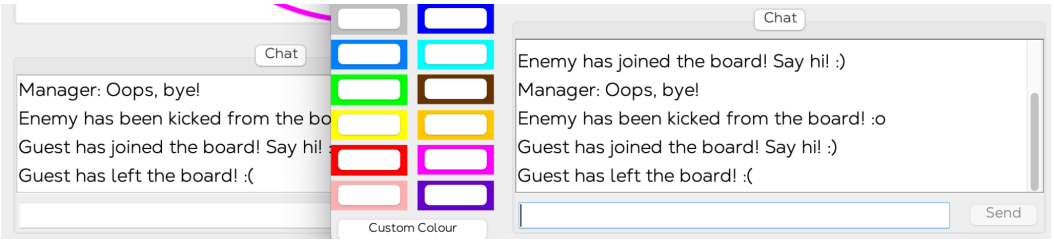
As specified above, when in the lobby, the users will be prompted to input a username. If found that the username has already been taken, the user will then be prompted to key in a different username.

4. Advanced Features

Advanced features implemented are listed below.

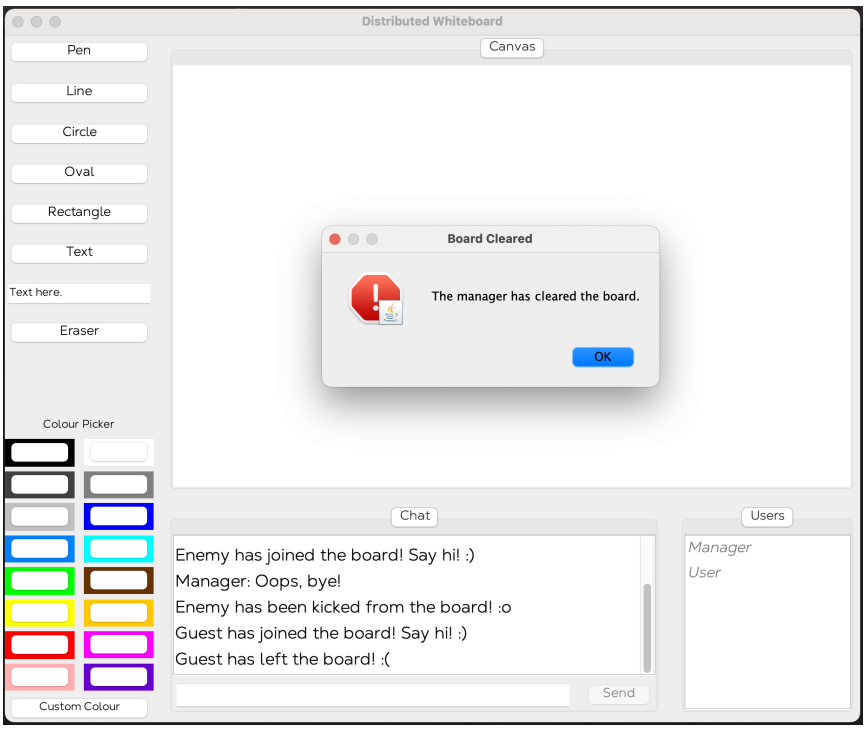
4.1. Text-Based Chat Box

Below the whiteboard canvas, a chat box is available for all users to use to send messages to each other. This chat box is also where users can see notifications when a user joins, leaves, or gets kicked out of the board.

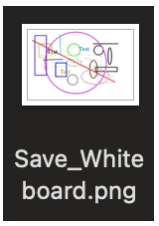


4.2. Whiteboard Creation

One of the manager-exclusive file methods is creating a new board. This clears the current board then broadcasts the cleared canvas to all the users.



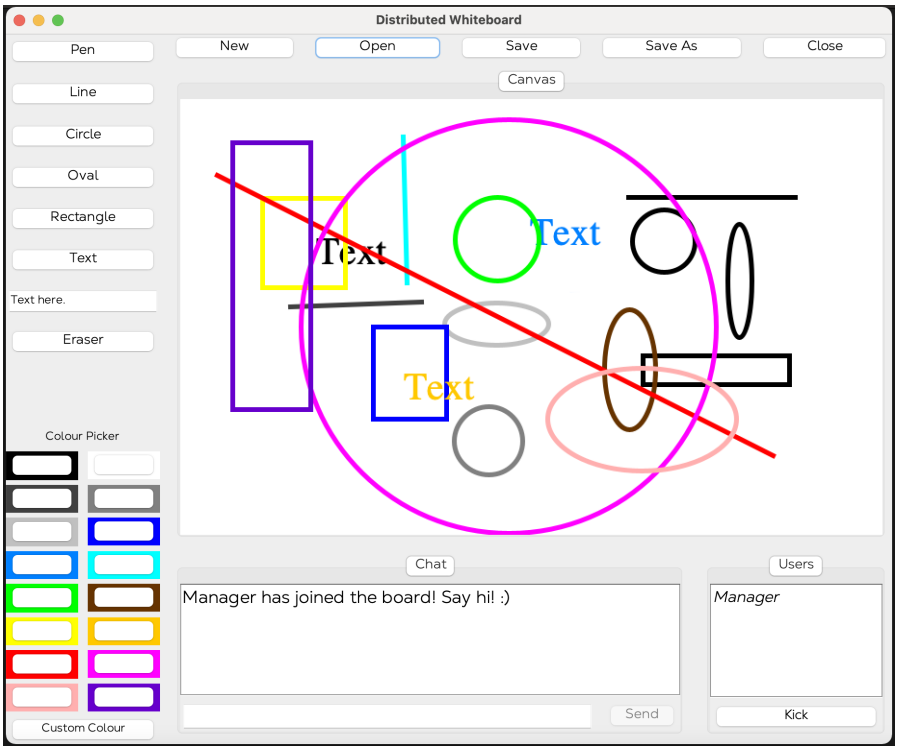
4.3. Whiteboard Saving



Managers can also save the canvas as a PNG or JPG file onto their devices at the location of their choosing. The system supports both Save and Save As features.

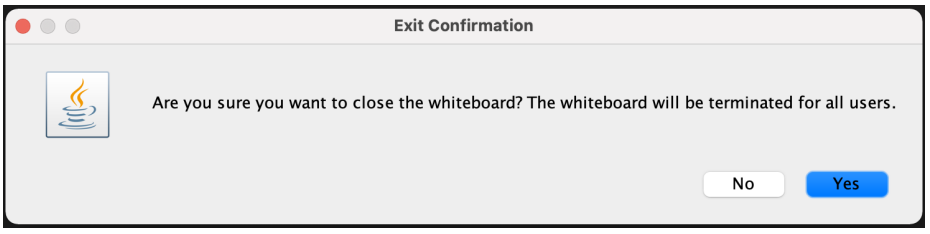
4.4. Saved Whiteboard Opening

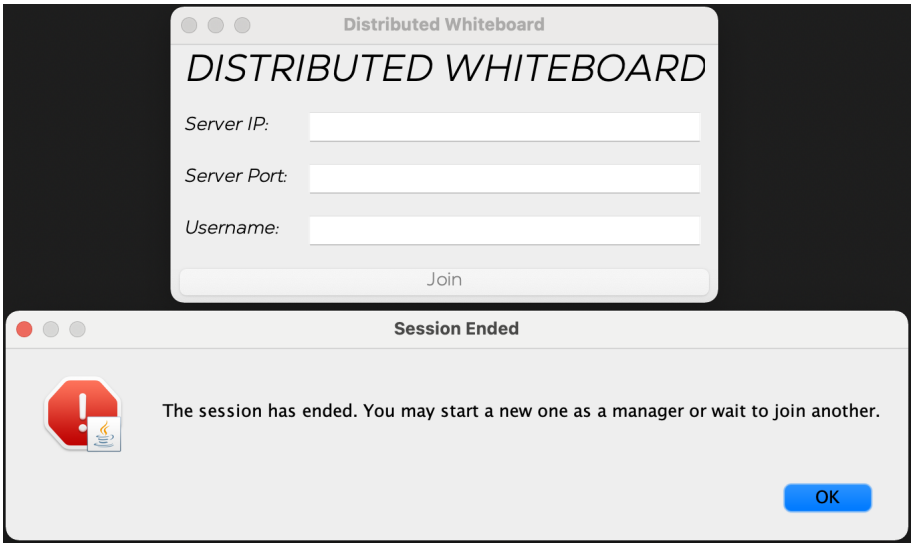
The above saved files can also be opened by the manager and this change will be broadcasted and propagated to all the users viewing the board.



4.5. Whiteboard Termination

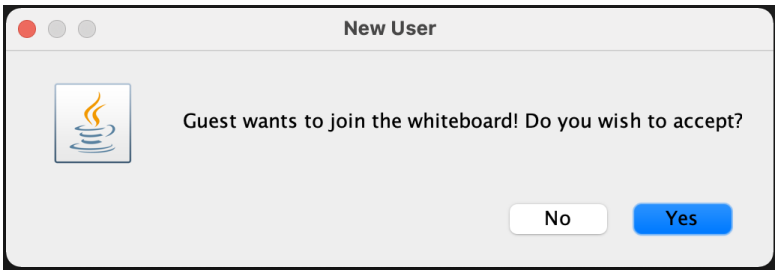
If the manager were to close the board, the application is then terminated for all users and all users will then be directed to the lobby where they can either create a new whiteboard as a new manager or wait to join another whiteboard as long as the servers remain running.

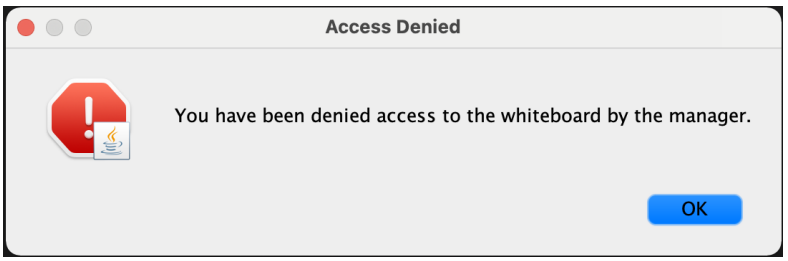




4.6. Approval-Based Joining

Upon signing up and before joining a board, managers will first be prompted to approve of the new user joining, and the new user will either be directed to the whiteboard (if approved) or will be notified of the manager's response (if denied).





4.7. Kicking Users

Managers also have the privilege of being able to kick any user they like. The kicked user is then notified of this and sent to the lobby.

5. Communication Protocol and Message Format

The system is built utilising the RMI protocol as the main form of communication between all the components of the system. This would mean that there is no need to plan for a specific format of messages to be sent to each other via an explicit TCP or UDP socket as RMI is object-oriented. This helps with scalability and complexity as we would also not need to bother with creation and maintenance of creating numerous threads to run the different processes from all the different components.

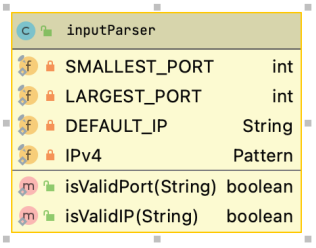
That said, the downside is that compared to a TCP socket, RMI demands a larger network overhead, which also means that the quality of the user's experience with the whiteboard would greatly depend on the network connection strength. RMI also requires all objects passed around be serialisable, which is the case for this whiteboard, but introduction of non-serialisable objects into the whiteboard system would then raise different issues.

6. Structure

This section describes the class structure and interactions of all the packages and classes used in the project.

6.1. Utilities Package

The `util` package consists only of the `inputParser` class. This is because it is more of a helper class that doesn't fit into any of the packages below. This class is only used for validation of the IP address and port number format.

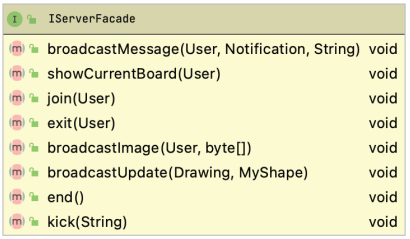


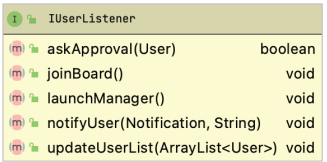
6.2. Remotes Package

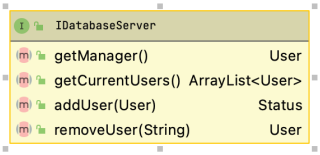
The `remotes` package defines all the interfaces of the classes that will be remotely accessed through the RMI protocol. This includes an interface for the following classes:

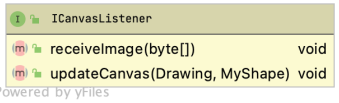
- `IDatabaseServer`
- `IServerFacade`
- `IChatListener`
- `ICanvasListener`
- `IUserListener`

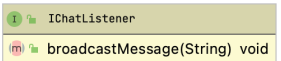
Since they are all interfaces, they do not have any dependencies. The methods defined here are just a subset of the actual methods that the actual classes may have. This is to allow abstraction to hide the complexity of the actual class as these are the only methods that the client and server need to and will be able to access. The other







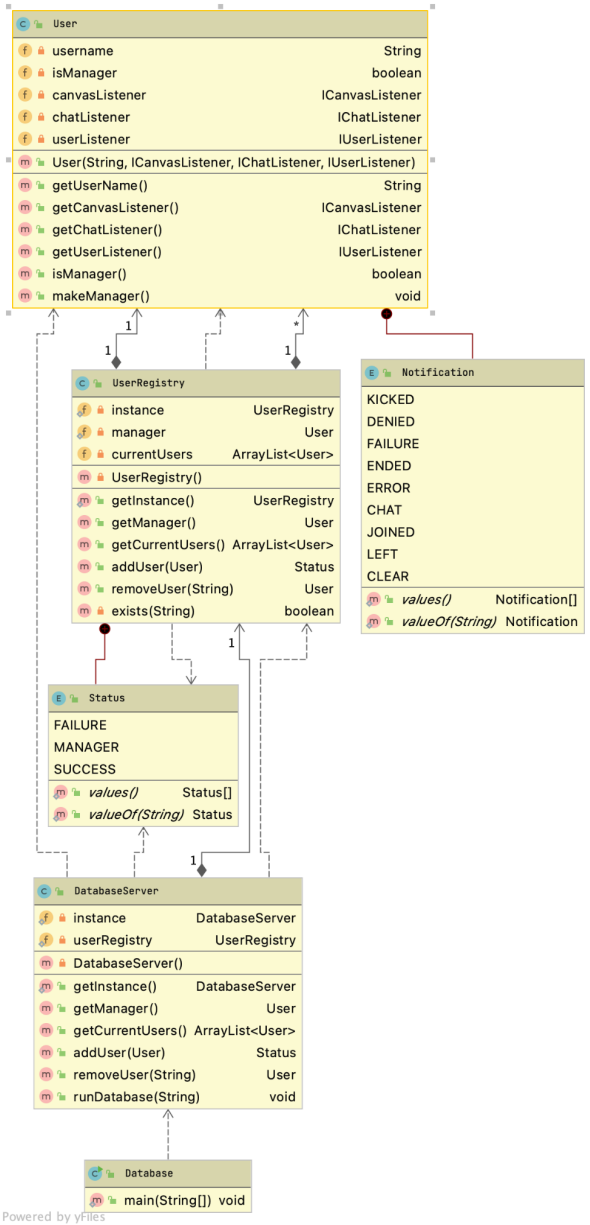




methods not specified would be called by classes within their respective packages.

6.3. Database Package

The `database` package represents a database system that will store a record of the current users accessing the whiteboard server. The role of the `Database` class is only to start the database server on a specified IP address and port number. This then results in the `DatabaseServer` class being instantiated along with a `UserRegistry` class that represents the user records. These classes are designed as Singleton classes as it would make sense that there is only one version of a record system (not considering replications with having numerous database servers in the real world). The DatabaseServer then creates the RMI registry on the specified port number and binds itself on the registry so that other components of the whiteboard system can access the specific methods.



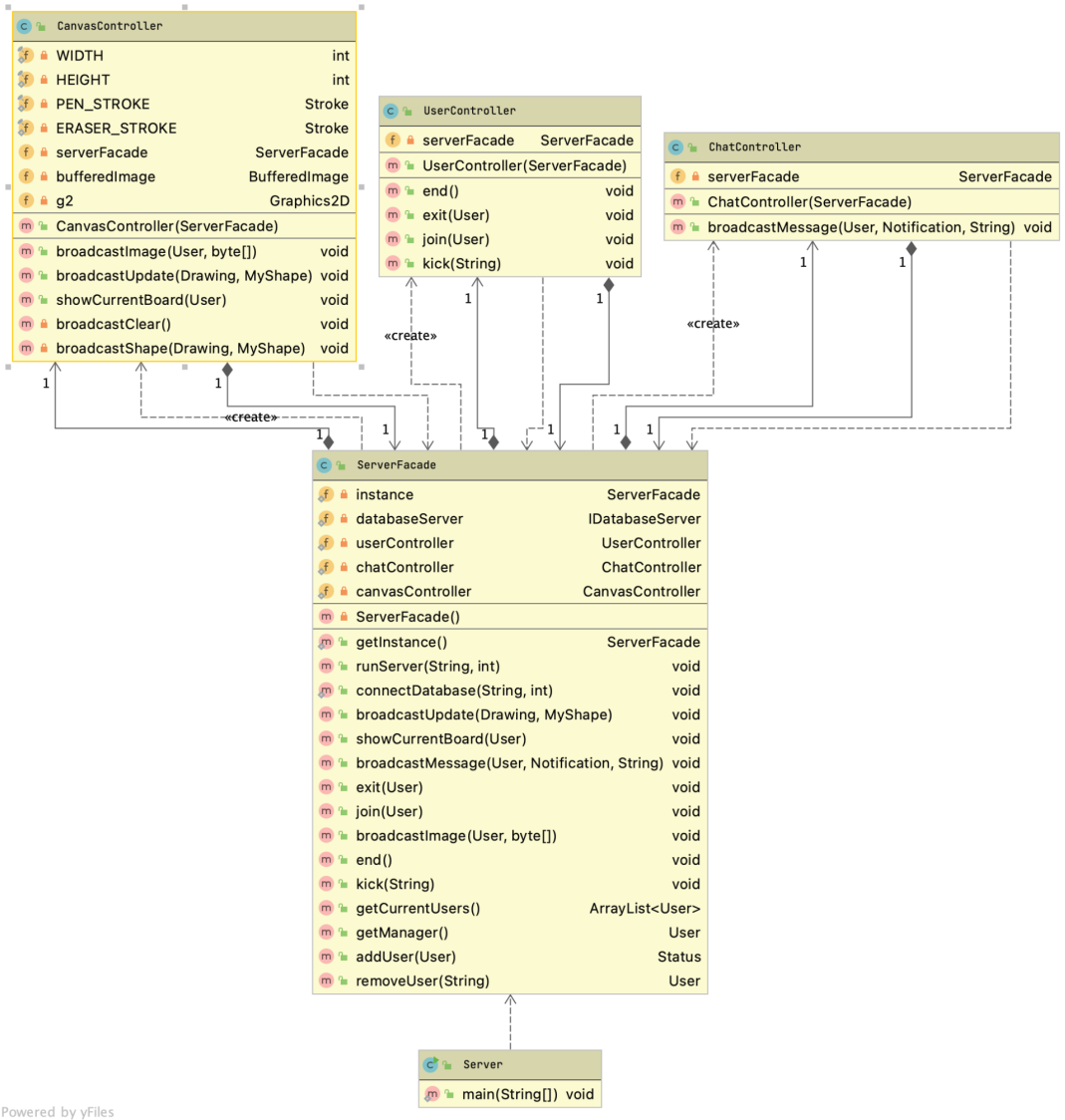
When a user signs up in the lobby, this would prompt the `DatabaseServer` to create a `User` class that will then be used to represent the user by the rest of the system. This `User` class is then passed around when the `DatabaseServer` calls the `UserRegistry` to try registering or removing the `User`. The `User` class will also contain some listener classes (see ***Section 6.5***).

6.4. Server Package

The `server` package represents the whiteboard server that will be hosting the whiteboard. Similar to the `Database` class, the `Server` class is only to start the whiteboard server on a specified IP address and port number. This then results in the `ServerFacade` class being instantiated. Since the specification requested that the whiteboard be of only a single instance that is shared among all the users, it would also make sense for the `ServerFacade` class to be a Singleton class so as to maintain a single whiteboard system state. On top of that, as the name suggests, the `ServerFacade` class follows the Facade design pattern as this is the only class that the client will be interacting with. This helps to hide the complexity of all the subcomponents within the system. This benefit is apparent in our system as the state of the whiteboard system is controlled using a variety of controller classes, i.e. the `CanvasController`, `UserController`, and `ChatController` classes.

The `CanvasController` class is responsible for updating the canvas state that all the whiteboards connected will show as well as broadcasting all changes to all the users. The `ChatController` class handles all the broadcasting of chat messages and notifications to users. Lastly, the `UserController` class is in charge of user management, such as facilitating connecting to and disconnecting from the whiteboard. Each of these controller classes will each be responsible for broadcasting their own updates to the users. Hence, they each have their own listener classes (see ***Section 6.5***) that they will be interacting with.

While it would make sense for the controller classes to also be singletons since the whiteboard system is to be a shared single state, leaving them as is would make it easier for modifications to the whiteboard server system in the future in the case that the whiteboard server is to host multiple whiteboards.



6.5. Client Package

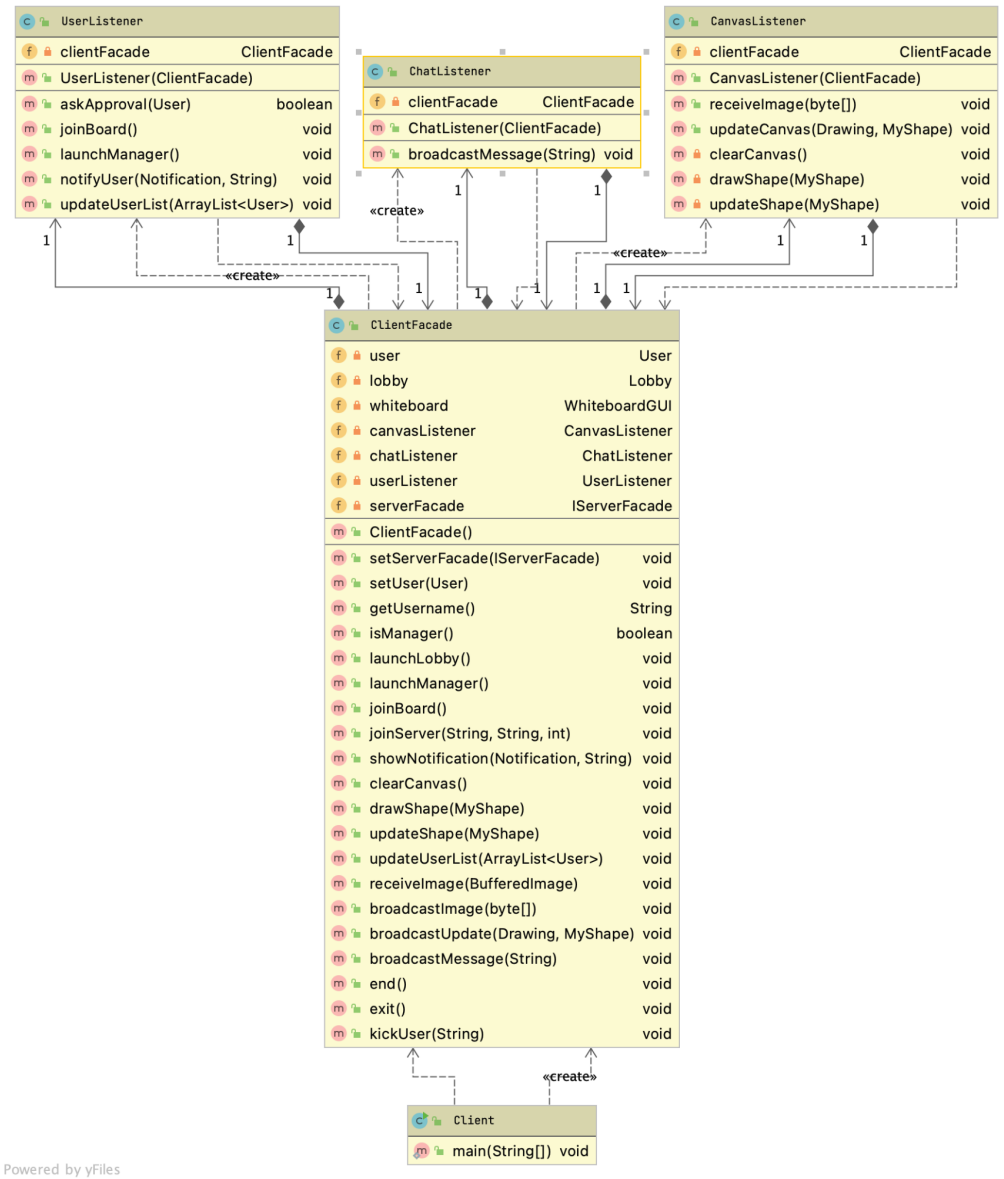
The `client` package represents the client application that the user will be interacting with. The design of this package is identical to the `server` package with the `Client` class being used only to start the `ClientFacade` class and launch the client application on the user's device. There are two main differences up to this point, first of which is that the `ClientFacade` class is not a singleton as different users would need to run their own client application. Secondly, the IP address and port number need not

be specified prior to the creation of the `ClientFacade` class as that is the role of the `gui` package (see ***Section 6.6***).

That said, it still adopts the Facade design pattern to hide the complexity of the cross communication of the classes. This benefit applies here because the `ClientFacade` class acts as the middleman between all the other components: the GUI, the server, and the listener classes.

Upon joining the board, the `ClientFacade` will look up the `ServerFacade` class on the RMI registry on the specified IP and port. If provided the right IP and port, and the `ServerFacade` class is found, the `ClientFacade` will then pass all the update information to `ServerFacade` class.

The listener classes – the `CanvasListener`, `UserListener`, and `ChatListener` classes – will then listen for responses that were broadcasted by the controller classes. These classes would then update the GUI through the ClientFacade class.



6.6. GUI Package

The `gui` package encompasses the whiteboard application that the user interacts with on their device. This can come in one of two forms: a `Lobby` class, or a `WhiteboardGUI` class.

The `Lobby` class is a standalone class that only has to handle the checking of the initial user inputs (IP, port, and username) and, if valid inputs, passes them to the `ClientFacade` that will then handle the joining of the whiteboard server. If unsuccessful in joining the server, the `Lobby` will persist and the user can keep submitting join requests until the user terminates the `Client` program.

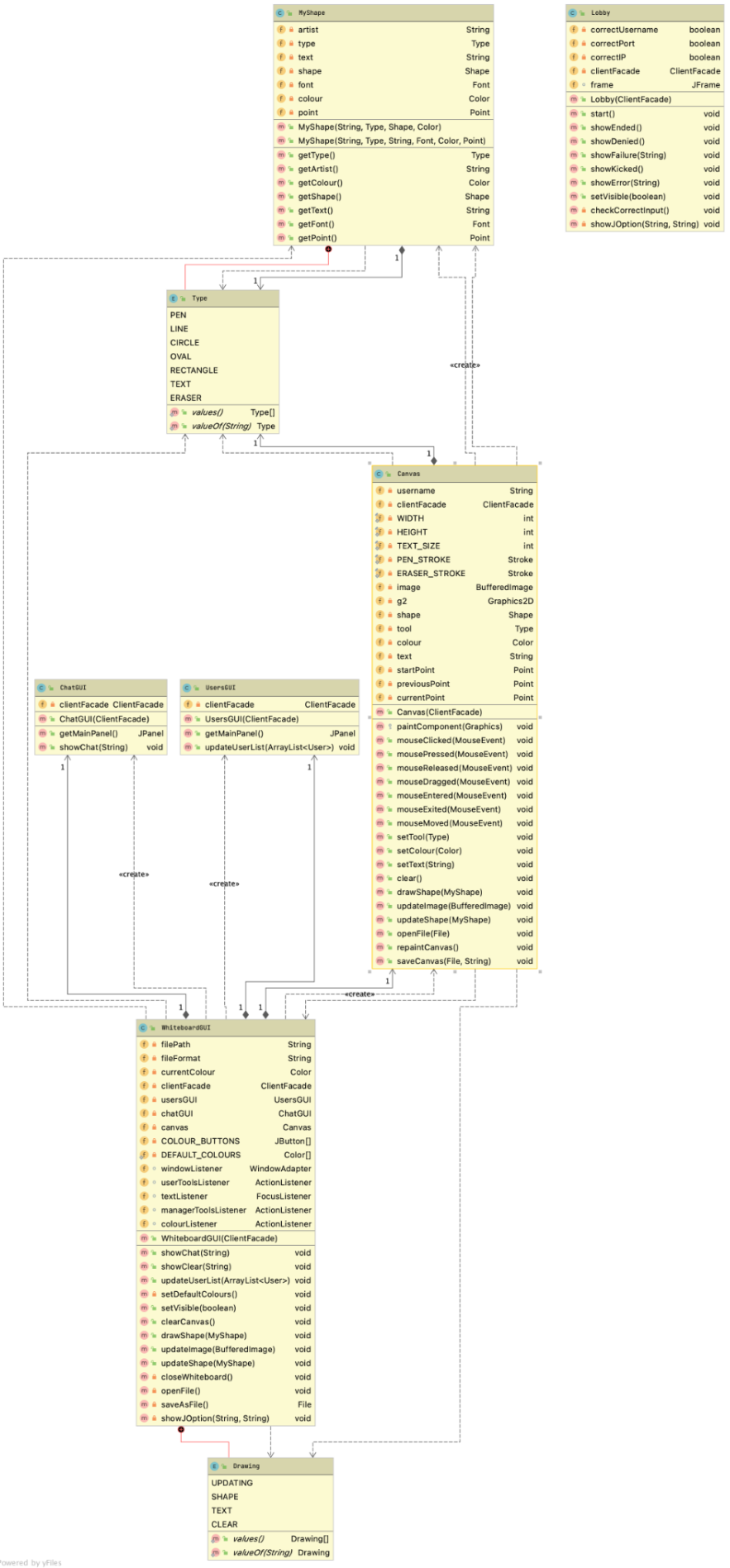
Upon joining the whiteboard server, the `WhiteboardGUI` class will take control of handling the rest of the user input and displaying all the whiteboard updates. The `WhiteboardGUI` class consists of the main application frame with all the tools, colour, and, if applicable, manager tools buttons. It will also contain a `Canvas` class, a `ChatGUI` class, and a `UserGUI` class as its components.

The `WhiteboardGUI` class will act as the middleman for the `ClientFacade` and these components as these components will send all the updates, shapes, and commands through the `WhiteboardGUI` class that will then pass it to the `ClientFacade` and so on to the whiteboard server. The `WhiteboardGUI` class will then be the one to receive the updates from the `ClientFacade` to call the right component to display said update.

The `Canvas` class component is the whiteboard canvas that the user can draw on upon selecting a tool and/or colour. It will be responsible for listening to the mouse events that the user performs on the canvas when drawing. It will then combine these mouse events into shapes in the form of a `MyShape` class. All the shapes drawn by the user and all the whiteboard updates received will be shown on the `Canvas` class.

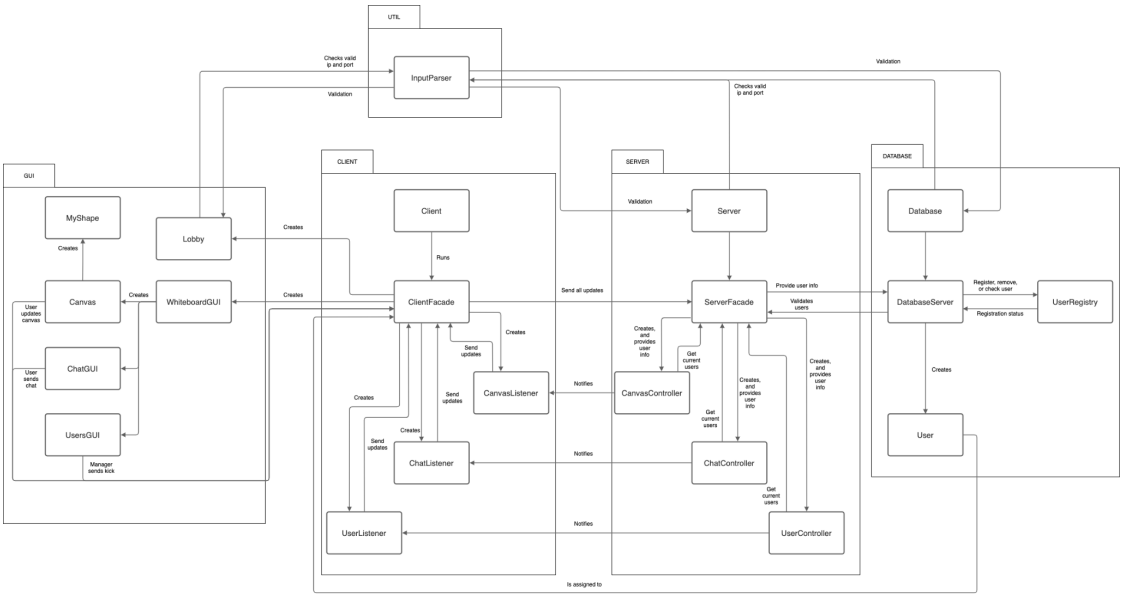
The `ChatGUI` class handles the chat messaging feature of the whiteboard application. This is where the user can input a text into the text box and send it out to other users. Chat messages as well as other user notifications such as when a user joins, leaves, or is kicked are all shown in the text area contained within this class.

Lastly, the `UserGUI` class is a component to show the list of active users that are editing the board. Besides that, this class is also the component that a manager will interact with when kicking a user.


MyShape
artist String
type Type
text String
shape Shape
font Font
colour Color
point Point
MyShape(String, Type, Shape, Color)
MyShape(String, Type, String, Font, Color, Point)
getType() Type
getArtist() String
getColour() Color
getShape() Shape
getText() String
getFont() Font
getPoint() Point
Lobby
correctUsername boolean
correctPort boolean
correctIP boolean
clientFacade ClientFacade
frame JFrame
Lobby(ClientFacade)
start() void
showEnded() void
showDenied() void
showFailure(String) void
showKicked() void
showError(String) void
setVisible(boolean) void
checkCorrectInput() void
showJOption(String, String) void
Type
PEN
LINE
CIRCLE
OVAL
RECTANGLE
TEXT
ERASER
values() Type[]
valueOf(String) Type
«create»
Canvas
username String
clientFacade ClientFacade
WIDTH int
HEIGHT int
TEXT_SIZE int
PEN_STROKE Stroke
ERASER_STROKE Stroke
image BufferedImage
g2 Graphics2D
shape Shape
tool Type
colour Color
text String
startPoint Point
previousPoint Point
currentPoint Point
Canvas(ClientFacade)
paintComponent(Graphics) void
mouseClicked(MouseEvent) void
mousePressed(MouseEvent) void
mouseReleased(MouseEvent) void
mouseDragged(MouseEvent) void
mouseEntered(MouseEvent) void
mouseExited(MouseEvent) void
mouseMoved(MouseEvent) void
setTool(Type) void
setColour(Color) void
setText(String) void
clear() void
drawShape(MyShape) void
updateImage(BufferedImage) void
updateShape(MyShape) void
openFile(File) void
repaintCanvas() void
saveCanvas(File, String) void
ChatGUI
clientFacade ClientFacade
ChatGUI(ClientFacade)
getMainPanel() JPanel
showChat(String) void
UsersGUI
clientFacade ClientFacade
UsersGUI(ClientFacade)
getMainPanel() JPanel
updateUserList(ArrayList<User>) void
«create»
«create»
«create»
WhiteboardGUI
filePath String
fileFormat String
currentColour Color
clientFacade ClientFacade
usersGUI UsersGUI
chatGUI ChatGUI
canvas Canvas
COLOUR_BUTTONS JButton[]
DEFAULT_COLOURS Color[]
windowListener WindowAdapter
userToolsListener ActionListener
textListener FocusListener
managerToolsListener ActionListener
colourListener ActionListener
WhiteboardGUI(ClientFacade)
showChat(String) void
showClear(String) void
updateUserList(ArrayList<User>) void
setDefaultColours() void
setVisible(boolean) void
clearCanvas() void
drawShape(MyShape) void
updateImage(BufferedImage) void
updateShape(MyShape) void
closeWhiteboard() void
openFile() void
saveAsFile() File
showJOption(String, String) void
Drawing
UPDATING
SHAPE
TEXT
CLEAR
values() Drawing[]
valueOf(String) Drawing
Powered by yFiles

6.7. Interaction Diagram

The following diagram displays the interaction between all the above components.



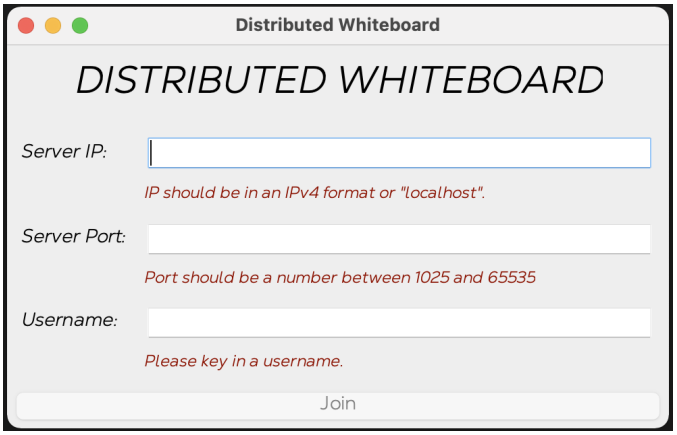
7. New Innovations

Below lists the extra aspects implemented past the requirements outlined in the specification.

7.1. Preliminary Error Handling

Preliminary checks for user input format and ensuring inputs are non-empty in the Lobby class component would help to avoid wasting computing power by the ClientFacade for trying to search for the WhiteboardServer on an invalid IP and port.

However, this does not mean that the client handles all the error handling – the database, whiteboard server, and client components would handle their own errors and print out meaningful error messages, be it through the terminal or through a popup.



7.2. User Database System

As mentioned above in ***Section 6.3***, the user database system helps to ensure that users can be identified and searched for via a unique username. While the unique username

identifier was part of the basic requirements, creating a database system was not. Having a database system separate from the whiteboard server would help to simulate real world scenarios where the server would not necessarily be hosted on the same machine as database servers.

7.3. Freehand Drawing

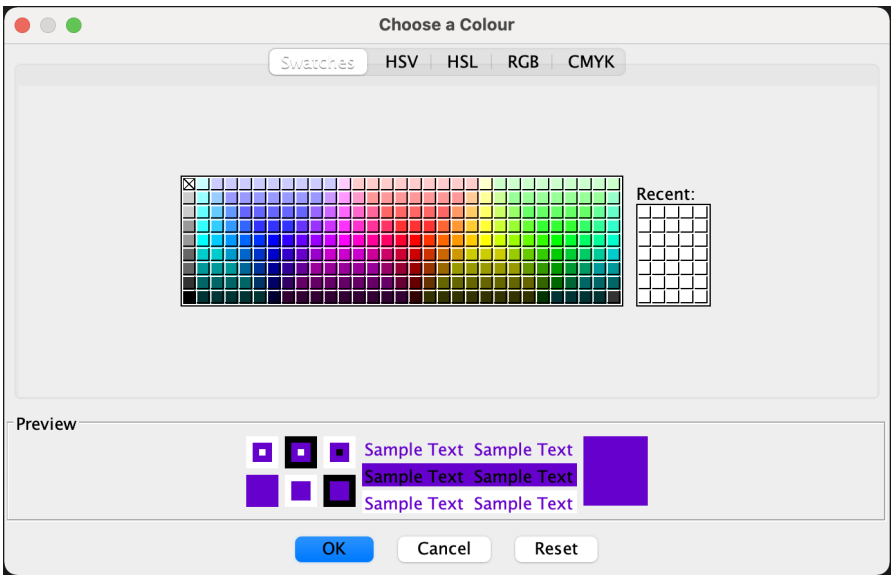
Users can use a Pen tool on the whiteboard application to draw shapes freehand.

7.4. Eraser Tool

Users can also use an Eraser tool to erase any part of the whiteboard.

7.5. Custom Colour Picker

Lastly, users can also create their own custom colours apart from the 16 provided colours using the custom colour picker. Users can create this custom colour by choosing from a larger selection of them, inserting the RGB values, and other methods.



8. Conclusion

In conclusion, this project provides a great preview as to difficulties and issues encountered when creating large-scale real-world applications. With the above implementations, this project has resulted in the successful creation of a stable and robust whiteboard system. One example proving the stability and robustness of the system is that if the server is down, the clients will be notified of the connection issue when trying to edit the whiteboard, and once it's back up, the users can join the whiteboard server again without having to restart the client whiteboard application.