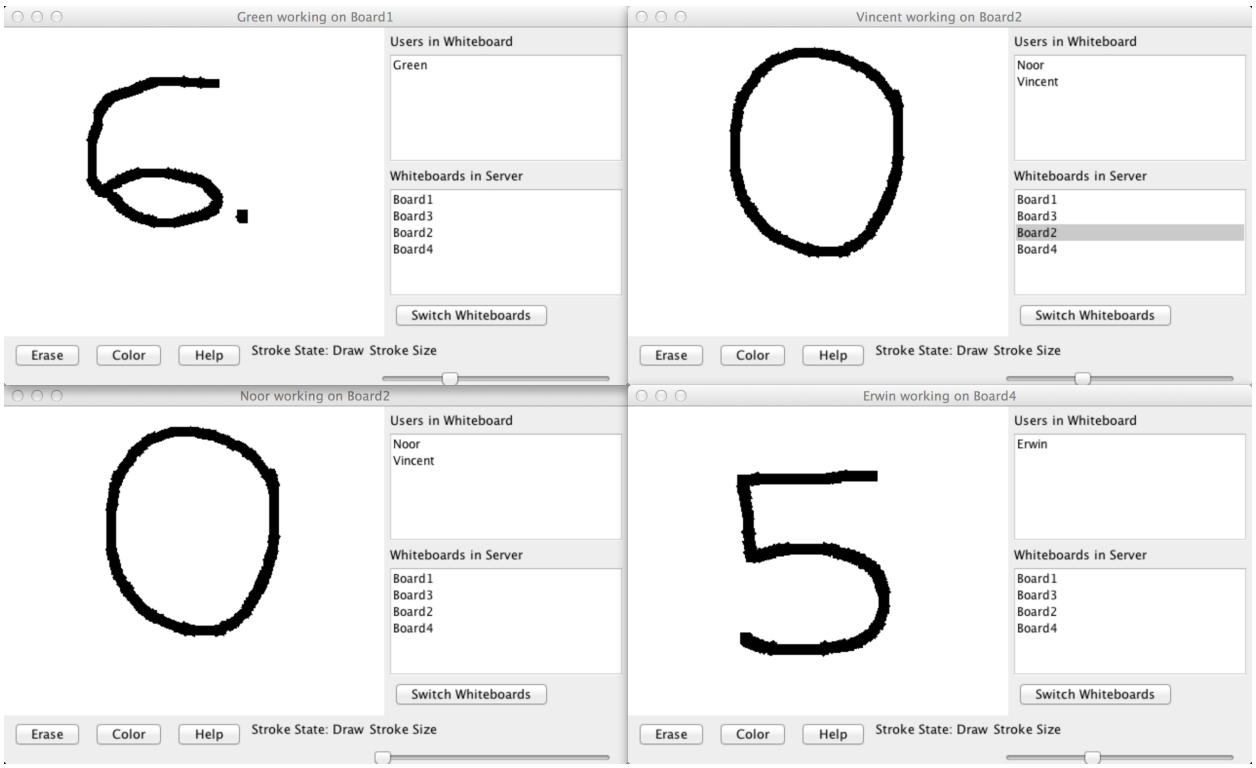
Design Document

Noor Eddin Amer
Erwin Hilton
Vincent Kee



Changes from Milestone

We have minimal functionality on the clients and servers
Noor talk about seperating client code here

General Control Flow

Server

1) The server initializes, creating a network socket and starting with three blank whiteboards.
2) At this point, the server is ready for any number of users to connect.

For each client:

1) The client starts up the GUI which has a pop-up window requesting a username. This username is sent to the server.

   The username must fulfill two conditions.

   a) It is unique. No two clients can have the same username.
   b) It is valid. It must have length greater than 0 and be composed of any characters except for spaces.
2) Once a valid username is entered, the user is prompted to select a whiteboard from the existing whiteboards or create a new whiteboard.
3) Now the Whiteboard graphical user interface loads with the state of the Whiteboard as stored on the server. The usernames of collaborators on the user's whiteboard are displayed in a list on the right side of the interface. The other Whiteboards in the server are displayed in a list on the right side of the interface.
4) The user can make any changes they want to the drawing canvas (drawing, erasing). All changes made by the user will be sent to all the other users working on the same Whiteboard as all changes made by other users on the Whiteboard will be propagated to the user. (Note that the ordering of edits is based on the order of the edits reaching the server).

   At any point the user can now

   a) Switch to another Whiteboard. The state of the current whiteboard will be preserved on the server with no effect on the other collaborators of the canvas. The new Whiteboard will be at the current state on the user's display, with all past changes present. The lists of displaying the collaborators on the Whiteboards will be updated.
   b) Disconnect from the server. The state of the whiteboard will be preserved on the server with no effect on the other collaborators of the canvas. The username will be discarded and the lists displaying the collaborators on the Whiteboards will be updated.

Client Server Model

A Whiteboard Client is the means by which a user can use a Whiteboard and collaborate with other users connected to the Whiteboard Server. The Whiteboard Client handles all incoming and outgoing messages being passed between the user and the server and calls the appropriate methods in the Whiteboard to update the user interface.

A Whiteboard Server is a server does all the message passing necessary for the users to collaborate using the Whiteboards over a network. In addition to handling incoming and outgoing messages, the server stores the history of all actions performed on each whiteboard so that new users can join whiteboards midway through a collaborative session without missing any information. Additionally, users can work on whiteboards over several network sessions as all information is stored on the server.

Class	**Constructor Signature (C) and Operations**
WhiteboardClient	**C:** public WhiteboardClient(Whiteboard whiteboard) **connectToServer():** Connects the client to the server socket and starts the threads to handle inputs and outputs between the server and the client. **handleServerResponse():** Listens to the socket for messages and passes inputs to the handleRequest(). **handleResponse():** Parses input from the server, performing appropriate operations on the Whiteboard class. **handleOutputs():** Polls the outputCommandQueue and writes items as text messages to the client's socket.
WhiteBoardServer	**C:** public WhiteboardServer(int port) **serve()**: Runs the server listening for client connections and handling them. Creates a Blocking Queue and adds it to the list of commandQueues in the index corresponding to the thread number. Calls createThreads(). **createThreads(final Socket socket, final Integer threadNum)**: Creates and starts threads to handle inputs and outputs between the server and specific client. **handleOutputs(final Socket socket, final Integer threadNum)**: Polls the output commandQueues of each client and writes items as text messages to the client's socket. **handleClientInput(final Socket socket, final Integer threadNum):** Listens to the client socket for messages and passes inputs to the handleRequest(). **handleRequest(final String input, final Integer threadNum):** Parses client input, performing appropriate operations. **runWhiteboardServer(final int port)**: Starts a WhiteboardServer running on the specified port.

Datatype Design

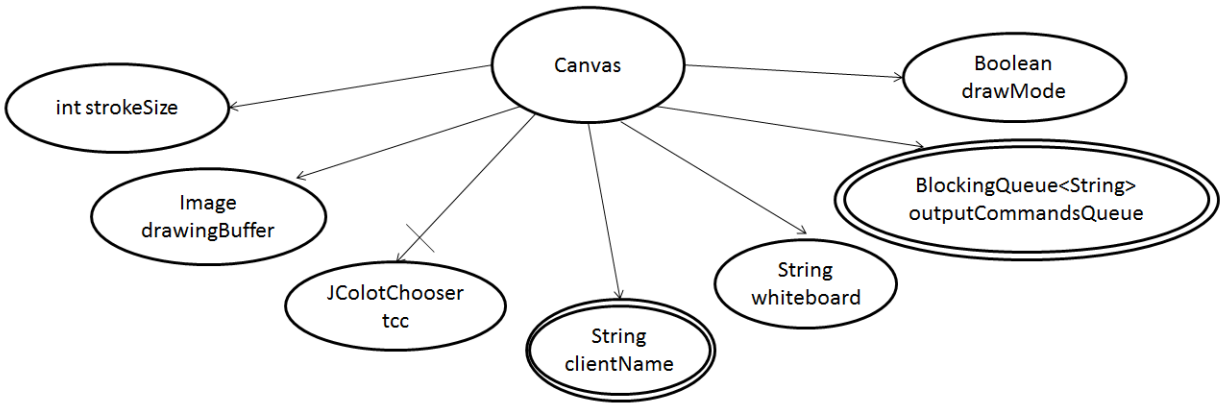
A Whiteboard is a user interface that contains an interactive drawing canvas along with a tool palette and various information displays. The interactive drawing canvas is a 2D array of pixels that can be manipulated by the user as well as any other users that have access to it through the Whiteboard server. The tool palette offers users the option of drawing in a variety colors, selecting stroke thickness, and even erasing lines. It also allows the user to switch to different drawing canvases if he or she desires to do so. The various information displays show the user information such as the other users working on the same canvas, messages between users working on the same canvas, and the other canvases on the server.

NOOR WRITEUP ABOUT HOW SEPERATED STUFF IN CLIENT MVC

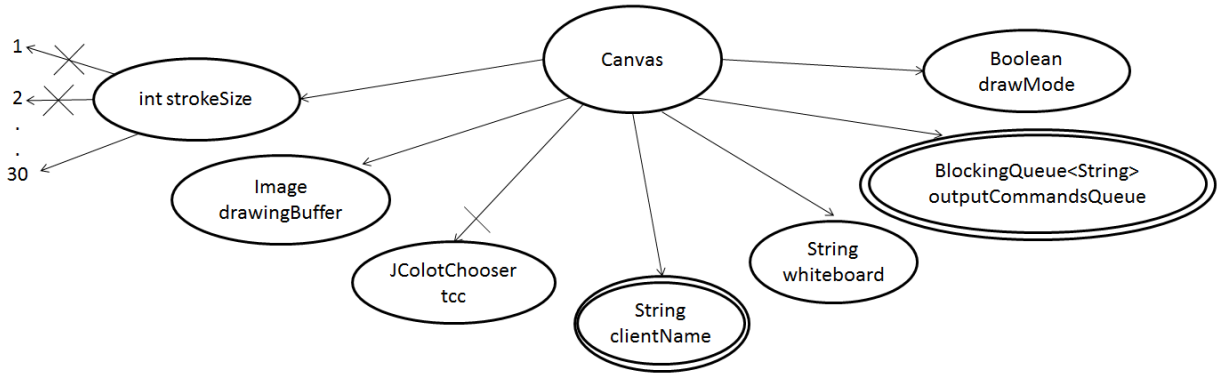
Class	**Constructor Signature (C) and Operations**
Canvas	**C:** public Canvas(int canvasWidth, int canvasHeight, BlockingQueue<String> queue) **getUsername()**: Pops up a window to get the username **paintComponent(Graphics g):** If there is no drawing buffer, it makes a drawing buffer. Otherwise, it copies the drawing buffer to the screen. **makeDrawingBuffer():** Make the drawing buffer and makes its a white blank canvas. **drawLineSegment(int x1, int y1, int x2, int y2):** Draws a line segment between two points (x1,y1) and (x2,y2) with a specified stroke size and color (in RGB), specified in pixels relative to the upper left corner of the drawing buffer **eraseLineSegment(int x1, int y1, int x2, int y2)**: Draws a white line segment between two points (x1, y1) and (x2, y2), specified in pixels relative to the upper-left corner of the drawing buffer. **commandDraw():** Draws a line segment between two points (x1,y1) and (x2,y2) with a specified stroke size and color (in RGB), specified in pixels relative to the upper left corner of the drawing buffer **commandErase():** Draw a white line between two points (x1, y1) and (x2, y2), specified in pixels relative to the upper-left corner of the drawing buffer. **addDrawingController()**: Adds the mouse listener that supports the user's freehand drawing. **getDrawingBuffer():** returns the drawingBuffer **getCommandQueue():** returns the commandQueue for th **makeCanvas()**: Sets up the User Interface **fillWithWhite()**: Makes the drawing buffer entirely white. **setStrokeState(int value)**: Sets the size of the stroke **checkRep():** * Checks that the rep invariant is maintained: * drawMode is true or false * height is greater than 0

	* width is greater than 0 * strokeSize is greater than or equal to 0 * tcc is not null * tcc has valid red, green and blue values * serverTcc is not null * serverTcc has valid red, green and blue values * whiteboard name is not null * whiteboard name is not an empty string * whiteboard name does not contain any spaces * outputCommandsQueue is not null * outputCommandsQueue length is 0 or more **Private Subclass DrawingController** **mousePressed(MouseEvent e)**: Starts drawing or erasing when the left mouse button is pressed **mouseDragged(MouseEvent e)**: Draws or erases a line segment when the the mouse moves while the left mouse button is pressed
WhiteboardGUI	**C:** WhiteboardGUI(int width, int height, BlockingQueue<String> outputCommandsQueue) **updateTitle()**: Updates the title of the JFrame GUI **createWindow()**: Creates a new window displaying the GUI - canvas, buttonPanel,sidePanel,etc **getExistingWhiteboards()**: Getter method for existingWhiteboards **getSidePanel()**: Getter method for sidePanel **getUsername()**: Gets the desired username from the user, checking against the existing usernames. Usernames CANNOT contain spaces or just be an empty string **chooseNewWhiteboard()**: Let's the user choose a new whiteboard. **helpBox()**: pops up helpBox with help message **colorChooser()**: pops up color chooser **makeWhiteboard()**: Uses an actionlistener on the to disconnect the client from the server

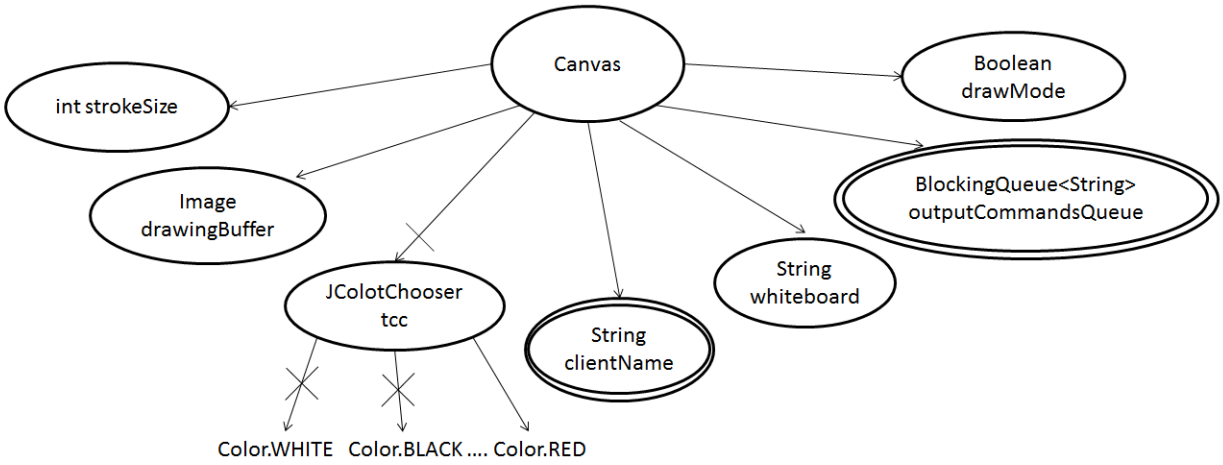
Snapshot Diagram of a Canvas in Action



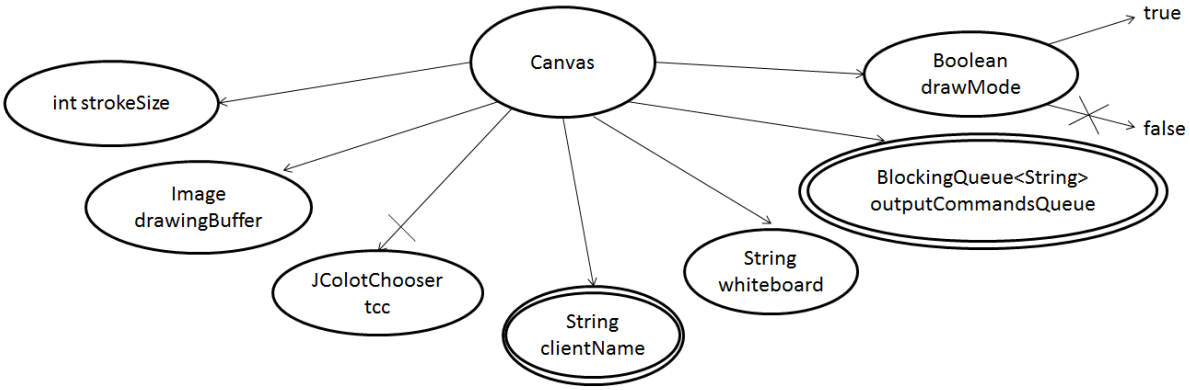
Changing Strokesize



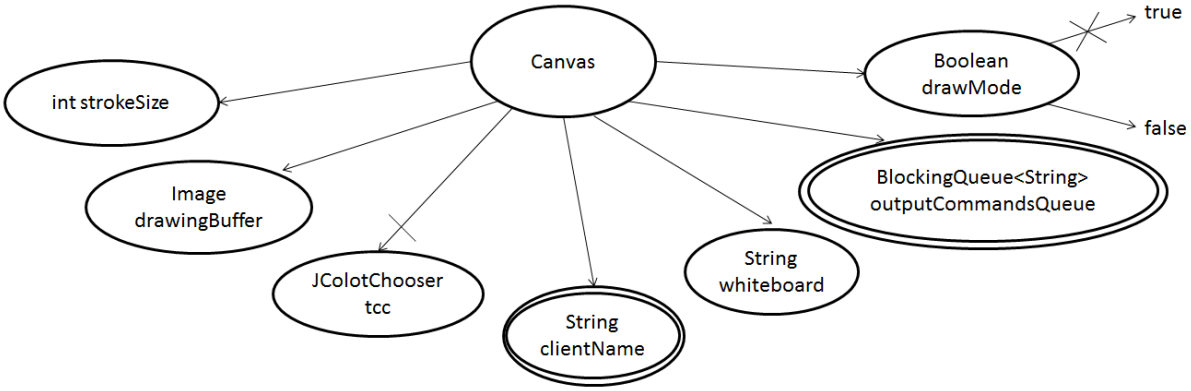
Changing Colors



Switching Erasing to Drawing (drawMode to true)

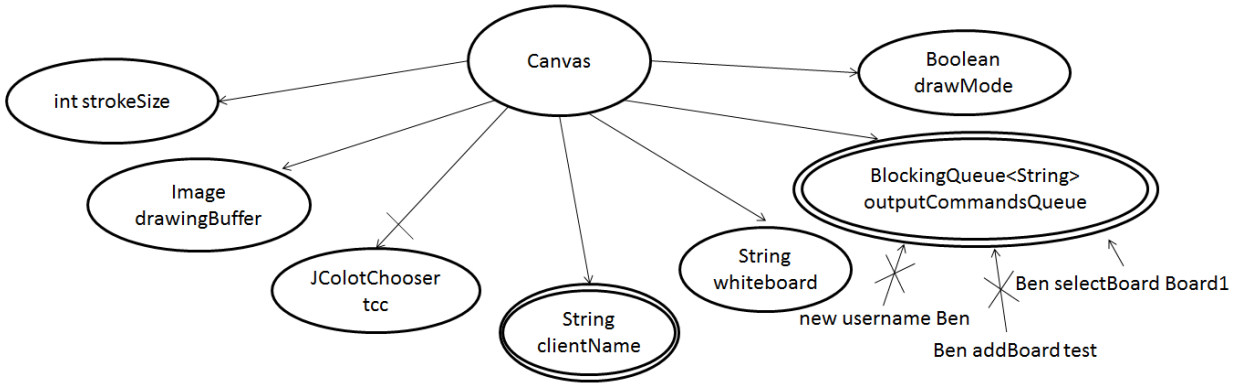


Switching Drawing to Erasing (drawMode to false)



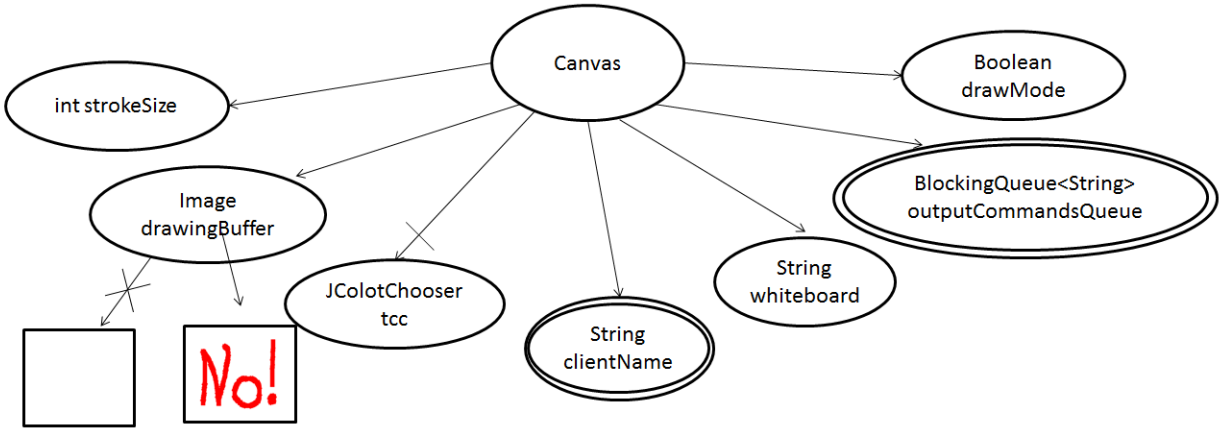
Offering to Queue

“Offering to Queue” : unsure about this one but, the immutable queue is offered different strings - so i guess it isn’t modified in itself? maybe doesn’t need its own bubble.

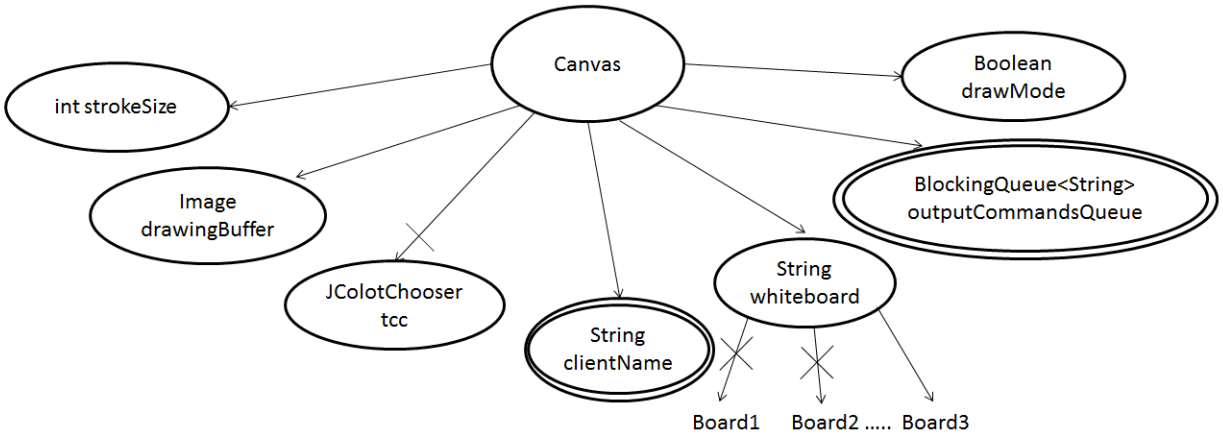


Drawing to Image

“Drawing to Image” Image drawingBuffer ...graphics drawn on? ignore this for now



Changing Whiteboard



Protocol

The Whiteboard Server and Client implement the client-server model. Each client passes text messages to the server and the server will parse the information, carry out the appropriate internal operations, and respond with text messages to all appropriate clients.

The server will have three threads running per client: one thread to listen for incoming text messages and place them into a BlockingQueue, one thread to poll the BlockingQueue of incoming messages, parse the commands, and carry out the appropriate operations (including placing outgoing text messages on another BlockingQueue), and one thread to poll the BlockingQueue of outgoing messages and write them to the socket.

Each client will be running three threads: one thread to listen for incoming text messages and place them into a BlockingQueue, one thread to poll the BlockingQueue of incoming messages, parse the commands, and carry out the appropriate operations (generally calling methods to update the GUI), one thread to run the GUI, and one thread to poll the BlockingQueue of outgoing messages and write them to the socket.

Grammar

Grammar for messages from the user to the server:

User-to-Server Whiteboard Message Protocol

MESSAGE :== (SELECTBOARD | DRAW | ERASE | HELP_REQ | BYE | NEWUSER | ADDBOARD) NEWLINE
SELECTBOARD :== CLIENT SPACE “selectBoard” SPACE BOARD
DRAW :== BOARDNAME SPACE “draw” SPACE X SPACE Y SPACE X SPACE Y SPACE STROKESIZE SPACE R SPACE G SPACE B
ERASE :== BOARDNAME SPACE “erase” SPACE X SPACE Y SPACE X SPACE Y SPACE STROKESIZE
CLIENT :== STRING
HELP_REQ :== CLIENT SPACE “help”
BYE :== CLIENT SPACE “bye”
NEWUSER :== “new” SPACE “username” SPACE USERNAME
ADDBOARD :== “addBoard” SPACE BOARDNAME
USERNAME :== STRING
BOARDNAME:== STRING
SPACE :== “ “
X :== INT
Y :== INT
R :== INT
G :== INT
B :== INT
STROKESIZE :== INT

```
INT             :== [0-9]+
STRING          :== [^=]*
NEWLINE         :== "\r?\n"
```

The action to take for each different kind of message is as follows:

SELECTBOARD Message:

1) If the username does not exist, add a message saying "Username does not exist." to the client's BlockingQueue in the commandsQueue list.
2) If the whiteboard does not exist, add a message saying "Whiteboard does not exist. Select a different board or make a new board." to the client's BlockingQueue in the commandsQueue list.
3) If the username and whiteboard exists, add the username and whiteboard to the HashMap clientToWhiteboardsMap, with the username as the key and the whiteboard as the value. Add a message saying "You are currently on board" (boardID) to the client's BlockingQueue in the commandsQueue list. Get the list of past commands on the whiteboard from the HashMap whiteboardToCommandsMap and add the contents to the client's BlockingQueue in the commandsQueue list.

DRAW Message:

1) If the whiteboard does not exist, return a message saying "Whiteboard does not exist." to the client.
2) If the drawing parameters are out of the boundaries of the canvas, return a message saying "Drawing parameters are out the drawing area of the whiteboard." to the client.
3) If the whiteboard exists and the drawing parameters are within the boundaries of the canvas, append the DRAW message to the ArrayList corresponding to the whiteboard key of the whiteboardToCommandsMap (Note that the client can only draw in the whiteboard that it is currently on - this is handled by the client as its drawing method will only put send the draw command with its whiteboard string id.) as well as the commandsQueue BlockingQueue.

ERASE Message:

1) If the whiteboard does not exist, return a message saying "Whiteboard does not exist." to the client.
2) If the erasing parameters are out of the boundaries of the canvas, return a message saying "Erasing parameters are out the drawing area of the whiteboard." to the client.
3) If the whiteboard exists and the erasing parameters are within the boundaries of the canvas, append the ERASE message to the ArrayList corresponding to the whiteboard key of the whiteboardToCommandsMap (Note that the client can only erase in the whiteboard that it is currently on - this is handled by the client as its erasing method will only put send the erase command with its whiteboard string id.) as well as the commandsQueue BlockingQueue.

HELP_REQ Message:

Adds a help message (see below) add the contents to the client's BlockingQueue in the commandsQueue list. Does not mutate any data structures.

BYE Message:

Adds the terminate message to the client's BlockingQueue in the commandsQueue list.

NEW USER Message:

1) If the user already exists, return a message saying "Username already taken. Please select a new username." to the client's BlockingQueue in the commandsQueue list.
2) If the user does not already exist and if there are no existing whiteboards, add the user as the key to the clientToWhiteboardsMap and with an empty String as a value. Also add the user as the key to the clientToThreadNumMap with the threadNum as the value. Add a message saying "No existing whiteboards" to the client's BlockingQueue in the commandsQueue list.
3) If the user does not already exist and there are existing whiteboards, add the user as the key to the clientToWhiteboardsMap and with an empty String as a value. Also add the user as the key to the clientToThreadNumMap with the threadNum as the value. For each whiteboard name, append it to a message starting with "Existing Whiteboards ". Add these messages to the client's BlockingQueue in the commandsQueue list. When there are no more whiteboard name messages, add a message saying "Done sending whiteboard names." to the client's BlockingQueue in the commandsQueue list.

ADD BOARD Message:

1) If the whiteboard already exists, add a message saying "Whiteboard already exists." to the client's BlockingQueue in the commandsQueue list.
2) If the whiteboard does not already exist, add the whiteboard name as the key to the whiteboardToCommandsMap with an empty ArrayList as a value. Add a message saying "Board " boardName "added" to the client's BlockingQueue in the commandsQueue list.

Grammar for messages from the server to the user:

Server-to-User Whiteboard Message Protocol

MESSAGE :== (SELECTBOARD | DRAW | ERASE | NEWUSER | HELP_REQ | BYE | ADDBOARD) NEWLINE

DRAW :== BOARD SPACE “draw” SPACE X SPACE Y SPACE X SPACE Y

ERASE :== BOARD SPACE “erase” SPACE X SPACE Y SPACE X SPACE Y

BOARD :== STRING

HELP_REQ :== “Instructions: username yourUsername, selectBoard board#, help, bye, board# draw x1 y1 x2 y2 strokesize R G B, board# erase x1 y1 x2 y2 strokesize ”

BYE :== “Thank you!”

ADDBOARD :== “No existing whiteboards.” | “Whiteboard does not exist. Select a different board or make a board.”

EXISTINGBOARDS :== “Existing Whiteboards” SPACE BOARDNAME

DONESENDING :== “Done sending whiteboards”

USERNAMETAKEN :== “Username already taken. Please select a new username.”

BOARDADDED :== “Board” SPACE BOARDNAME SPACE “added”

BOARDNAME:== STRING

SPACE :== “ “

X :== INT

Y :== INT

INT :== [0-9]+

STRING :== [^=]*

NEWLINE :== “\r?\n”

The action to take for each different kind of message is as follows:

USERNAMETAKEN Message:

Creates a popup window to get the username from the user. Adds the message “new username desiredClientName” (where desiredClientName is the username entered by the client) to the outputCommandsQueue.

ADDBOARD Message:

Creates a popup window to get a new whiteboard name from the user. Adds the message “addBoard desiredWhiteboardName” (where desiredWhiteboardName is the whiteboard name entered by the client) to the outputCommandsQueue.

EXISTINGBOARDS Message:

Adds the whiteboard name to the list of existing whiteboards.

DONESENDING Message:

Creates a popup window that lets the user choose a whiteboard or create a new one.

BOARDADDED Message:

Sets the boardname specified in the Message as the whiteboard name of the client.

DRAW Message:

Draws the line segment defined in the message.

ERASE Message:

Erases the line segment defined in the message.

HELP_REQ Message:

Creates a popup window with help instructions.

BYE Message:

Terminates the connection with the server.

Concurrency Strategy

Thread Safety Argument for the Whiteboard Server
All fields in the Whiteboard Server are final and private or protected (the only other class in the package is the class for tests) so there is no representation exposure. The commandQueues is a Synchronized Arraylist of BlockingQueues, each of which contains the messages to be sent to its unique client. Each client socket is mapped to a threadNum which is an Integer derived from an Atomic Integer counter that keeps track of which client socket belongs to which client. This threadNum also corresponds to the client's BlockingQueue in the commandQueues list. This ensures that the correct messages are always sent to the correct client. The socket and threadNum are passed between threads as final objects to prevent mutation. The assignments of clients to whiteboards, whiteboards to their command history, and client to threadNum are all kept track of with Synchronized HashMaps, which prevent race conditions as different threads access the information to properly distribute information to the correct clients. The outputThreadActive Synchronized list keeps track of the status of all the clients. This allows the server to terminate its output threads once the client disconnects. All other variables (incoming and outgoing messages) are kept local to maintain thread confinement. This allows us to avoid the use of locks, removing the risk of any deadlocks. Therefore, the Whiteboard Server is thread-safe.

Thread Safety Argument for the Whiteboard Client
All fields in the Whiteboard Client are private or protected so there is no representation exposure. Like in the Whiteboard Server, the outputCommandsQueue is a Blocked Queues so that there will be no race conditions as the threads access them. The fields that are not final, the clientName and the whiteboard name, can only be modified in one thread so they are threadsafe by thread confinement. All other variables (incoming and outgoing messages) are kept local to maintain thread confinement. This allows us to avoid the use of locks, removing the risk of any deadlocks. Therefore, the Whiteboard Client is thread-safe.

Thread Safety Argument for the WhiteboardGUI
All fields in the Whiteboard GUI are private or protected so there is no representation exposure. The Whiteboard is only accessed by one thread on the Whiteboard Client (the thread that parses commands from the BlockingQueue of commands from the server). Therefore, there is no risk of race conditions, as changes to the mutable fields are confined to one thread (thread confinement). As a result, no locks were used on the WhiteboardGUI. Therefore, there is no risk of deadlock on the WhiteboardGUI. Therefore, the WhiteboardGUI is thread-safe.

Thread Safety Argument for the Canvas
All fields in the Canvas are private or protected so there is no representation exposure.The Canvas is accessed by one thread on the Whiteboard Client (the thread that parses commands from the BlockingQueue of commands from the server) and the thread from the GUI. There is no risk of race conditions, as all fields except for the drawingBuffer are accessed by only one

thread. The drawingBuffer always reflects the state in the server, so there are no risk of race conditions. As a result, no locks were used on the Canvas. Therefore, there is no risk of deadlock on the Canvas. Therefore, the Canvas is thread-safe.

Thread Safety Argument for the SidePanel
All fields in the SidePanel are private or protected so there is no representation exposure. The SidePanel is only accessed by one thread on the Whiteboard Client (the thread that parses commands from the BlockingQueue of commands from the server). Therefore, there is no risk of race conditions, as changes to the mutable fields are confined to one thread (thread confinement). As a result, no locks were used on the SidePanel. Therefore, there is no risk of deadlock on the SidePanel. Therefore, the SidePanel is thread-safe.

Thread Safety Argument for the ButtonPanel
All fields in the ButtonPanel are private or protected so there is no representation exposure. The ButtonPanel is only accessible by the Swing event handler thread. Therefore, there is no risk of race conditions, as changes to the mutable fields are confined to one thread (thread confinement). As a result, no locks were used on the ButtonPanel. Therefore, there is no risk of deadlock on the ButtonPanel. Therefore, the ButtonPanel is thread-safe.

Therefore, our implementation is thread-safe.

Testing Strategy

All operations of the classes in both the Server, Client, and Whiteboard can and should be tested for correctness. Since this project includes a graphical user interface that requires user response at various stages and a text-based message passing protocol, full testing coverage would be hard to achieve without using manual tests. In addition to manual tests, we will be using JUnit tests to test our drawing canvas and the protocol on the server. Therefore, we will conduct both manual and JUnit tests to thoroughly test our implementation and ensure correctness.

We will conduct our tests as follows:

1. Canvas (Drawing Canvas Datatype Definition)
In testing our Whiteboard canvas, we will use both JUnit and manual tests.

Our checkRep() method will be run in each test, asserting that: drawMode is true or false, height is greater than 0, width is greater than 0, strokeSize is greater than or equal to 0, tcc is not null, tcc has valid red, green and blue values, serverTcc is not null, serverTcc has valid red, green and blue values, whiteboard name is not null, whiteboard name is not an empty string, whiteboard name does not contain any spaces, outputCommandsQueue is not null, and outputCommandsQueue length is 0 or more.

Our methods in canvas will be tested as follows:

Features Tested	**Operations Tested**	**Testing Strategy (Partition of the I/O Space and Expected Behavior)**

makeDrawingBuffer() creates a new drawingBuffer, and calls fillWithWhite. Check to see if all the pixels non null
fillWithWhite()/eraseBoard() will be tested by checking to make sure all the pixels are filled with Color.WHITE

setStrokeState() called by GUI slider. Check this manually using the GUI
drawLineSegment()- test the output strings (correct x1,y1,x2,y2 coordinates and rgb values)
eraseLineSegment() test output strings (correct x1,y1,x2,y2 coordinates, rgb value should be

0,0,0 (white)).
test strings sent out by drawLineSegment and eraseLineSegment (what goes to outputQueue)
test the commandDraw and commandErase (what is drawn on canvas) - test different rgb strings and make sure the drawLineSegment and eraseLineSegment methods are called, also make sure the color chooser is set to the correct color according to the input rgb strings
color changed by GUI color chooser - after setting the color chooser to a specific color (using color chooser.setColor or the commandDraw command) - check to make sure it is the right color

Manual testing for full functionality of the Whiteboard canvas will be done in the end to end tests as discussed below.

2. Server Protocol Testing (JUnit tests)
The WhiteboardServerTest.java runs JUnit tests on components of the Server that do not require sockets to function properly. It also tests the proper parsing of the message passing protocol on the Server end (with single user as multiple users are tested in the end to end tests).

Features Tested	**Operations Tested**	**Testing Strategy** **(Partition of the I/O Space and Expected Behavior)**
Server Autocreating Boards	WhiteboardServer createBoards()	**I/O Space Partitioning** No partitions as method generates the predefined boards **Expected Behavior** Three Boards named “Board1”, “Board2”, and “Board3” are created and put in the whiteboardToCommandsMap.
Message Parsing Protocol	WhiteboardServer handleRequest() WhiteboardServer getExistingWhiteboards All() Whiteboard Server getExistingWhiteboards One(threadNum)	**I/O Space Partitioning** Input - new username that doesn’t exist, new username that exists, add whiteboard that exists, add whiteboard doesn't exist, select whiteboard with no user but with a board, select whiteboard with no board but with a user, select whiteboard with normal draw single user, erase with single user, command in regex but no specified action, command not in regular expression **Expected Behavior** See Protocol

3. Manual End-to-End Testing (Server, Client, GUI)
We used manual end to end tests to test our server, client, and GUI’s functionality and performance as various phases required user input to be able to test thoroughly.

For each feature tested, we specified the operations tested as well as the testing strategy and expected behavior as displayed in the table below:

Features Tested	**Operations Tested** (Note that only the specific method being tested is listed, main methods and other methods that start the client and server are implicitly tested)	**Testing Strategy (Partition of the I/O Space and Expected Behavior)**
	The following 3 features occur at the start of a client joining the server and so occur sequentially	
1. Client Connecting to Server	WhiteboardServer main() WhiteboardServer runWhiteboardServer(final int port) WhiteboardClient main(String[] args) WhiteboardClient runWhiteboardClient (final String ipAddress, final int port, final int clientWidth, final int clientHeight) WhiteboardClient connectToServer()	**I/O Space Partitioning** IP Address - 3 different computers as server Ports - 5 different ports **Expected Behavior** Client able to connect to server with different IP addresses and ports (with proper arguments entered by the client). No username conflicts with simultaneous user submissions as the username is granted to the client that connects to the server first.
2. Username Dialog Box	WhiteboardServer handleClientInput(final Socket socket, final Integer threadNum) WhiteboardServer handleRequest(final String input, final Integer threadNum) WhiteboardServer handleOutputs(final Socket socket, final Integer threadNum)	**I/O Space Partitioning** Valid Username - Length 1 string, length 10 string, string with special characters, string with various capitalizations, length 20 string Invalid Username - empty string, space, string with any number of spaces interleaved with characters Valid Usernames are defined as all strings of length greater than 0 composed of all possible characters except for spaces. Invalid Usernames are defined as an empty string or a string containing any number of spaces.

	WhiteboardClient handleServerResponse(String input) WhiteboardClient createGUI() WhiteboardGUI getUsername(String message)	**Expected Behavior** Username dialog box first pops up with a message saying “Input your desired username:” and a textfield for the user to enter input. The textfield should have the highlighted text “username”. The user enters his/her input by either hitting “ENTER” after typing in their username or clicking the OK button. We have not built in support for clicking the X or Cancel button as Professor Goldman instructed us to focus on Whiteboard functionality rather than making the GUI bulletproof to users trying to break the program. If the user enters an invalid username, a new dialog box appears saying “Please enter a username with no spaces composed of at least 1 character. \n Please input a valid username:” and a textfield for the user to enter input. Note that this dialog box will loop until the user selects a valid username. If the username enters a username that is already taken by another client on the server, a new dialog box appears saying “Username already taken. Input your desired username:”. Note that this dialog box will loop until the user selects a unique username. Once a valid unique username has been selected, a whiteboard selection box should appear.
3. Whiteboard Dialog Box	WhiteboardServer handleClientInput(final Socket socket, final Integer threadNum) WhiteboardServer handleRequest(final String input, final Integer threadNum) WhiteboardServer handleOutputs(final Socket socket, final Integer threadNum) WhiteboardClient handleServerResponse(String input) WhiteboardClient createGUI()	**I/O Space Partitioning** Valid Whiteboard Name - Length 1 string, length 10 string, string with special characters, string with various capitalizations, length 20 string Invalid Whiteboard Name - empty string, space, string with any number of spaces interleaved with characters Valid Whiteboard Names are defined as all strings of length greater than 0 composed of all possible characters except for spaces. Invalid Whiteboard Names are defined as an empty string or a string containing any number of spaces. **Expected Behavior** Username dialog box first pops up with a message saying “Existing Whiteboards: ” followed by the names of the Whiteboards on the server. On the

	WhiteboardClient createWhiteboard() WhiteboardGUI chooseWhiteboardPopup()	next line, there will be a message saying “Enter the name of an existing whiteboard or type in a new whiteboard name” and a textfield for the user to enter input. The user enters his/her input by either hitting “ENTER” after typing in their username or clicking the OK button. We have not built in support for clicking the X or Cancel button as Professor Goldman instructed us to focus on Whiteboard functionality rather than making the GUI bulletproof to users trying to break the program. If the user enters an invalid whiteboard name, a new dialog box appears saying “Please enter a whiteboard name with no spaces composed of at least 1 character. \n Please input a valid whiteboard name:” and a textfield for the user to enter input. Note that this dialog box will loop until the user selects a valid whiteboard name. Once the valid whiteboard is selected, the GUI should be fully visible to the user with the title displaying the username followed by “working on” whiteboard name.
	The following features can occur in any order depending on the users’ behavior	
4. Side Panel	WhiteboardServer handleRequest(final String input, final Integer threadNum) WhiteboardServer handleOutputs(final Socket socket, final Integer threadNum) SidePanel updateWhiteboardsList(List<String> whiteboards, String currentWhiteboard) SidePanel updateClientsList(List<String> clients)	**I/O Space Partitioning** Whiteboards on Server - 0 starting boards, 1 starting board, 3 starting boards, 20 starting boards (more than directly visible in the Whiteboards in Server JList) Users Working on the Same Whiteboard - 0 collaborators, 1 collaborator, 20 collaborators (more than directly visible in the Users in Whiteboard JList) **Expected Behavior** The Users in Whiteboard displays all the other users working on the same Whiteboard. As soon as users log off or switch boards, the display should be updated immediately. If there are more users working on the same Whiteboard then there is vertical room in the display, a vertical scroll bar

		should appear that allows the user to scroll through and view all collaborators. The Whiteboards in Server displays all the other Whiteboards on the server. As soon as a new board is added, the display should be updated immediately. If there are more whiteboards on the same server then there is vertical room in the display, a vertical scroll bar should appear that allows the user to scroll through and view all whiteboards. Users can select a Board from the list and click the “Switch Whiteboards” button located right below the Whiteboards in Server display to switch to the Whiteboard. The users’ drawing canvas should be erased clean and then redrawn with the new Whiteboard history immediately. The drawing canvas should reflect the same image as all the other users working on the same Whiteboard. The user’s pen mode (Draw or Erase), stroke size, and color should be preserved between changes in Whiteboards.
5. Drawing Canvas	Canvas paintComponent(Graphics g) Canvas makeDrawingBuffer() Canvas fillWithWhite() Canvas drawLineSegment(int x1, int y1, int x2, int y2) Canvas eraseLineSegment(int x1, int y1, int x2, int y2) Canvas commandDraw(int x1, int y1, int x2, int y2, int currentStrokeSize, String red, String green, String blue) Canvas commandErase(int x1, int y1, int x2, int y2, int newStroke) Canvas	**I/O Space Partitioning** Draw - Drawing with various stroke sizes and colors as the only user, with one other user, with three other users, with more than three users Erase - Erasing with various stroke sizes as the only user, with one other user, with three other users, with more than three users **Expected Behavior** As users draw and erase on their drawing canvases, the changes should be propagated across all users working on the same whiteboard (with minimal lag, subject to network speeds) in the order that the actions are received by the server. Therefore, the same image should be present across all collaborators once all commands are propagated to all collaborators.

	addDrawingController() Canvas checkRep()	
6. Button Panel	Canvas drawLineSegment(int x1, int y1, int x2, int y2) Canvas eraseLineSegment(int x1, int y1, int x2, int y2) Canvas commandDraw(int x1, int y1, int x2, int y2, int currentStrokeSize, String red, String green, String blue) Canvas commandErase(int x1, int y1, int x2, int y2, int newStroke) Canvas setStrokeState (int value) Canvas getTcc() WhiteboardGUI helpBox()	**I/O Space Partitioning** Draw Mode - Draw, Erase Stroke Size - Smallest, Middle, Largest Size Colors - Different colors from each of the tabs in the Color Chooser User clicks the help button **Expected Behavior** Users can click on the leftmost bottom button which toggles between switching the pen to “Draw” and “Erase” modes. The text label beginning with “Stroke State: ” should accurately display the mode of the pen, which is the opposite of what is displayed on the button as the button changes the state. Across changing states to Draw or Erase, the color of the pen in Draw Mode should not change. However, the stroke size of the pen remains whatever it was in the previous state (Draw and Erase mode share stroke size data). Users can drag the slider on the bottom right hand corner to change the stroke size of the pen. Users can click on the Color button which displays a Color Chooser pop-up window allowing them to select a color for their pen. They can select from different color models including Hue, Saturation, Value (HSV), Hue, Saturation, Lightness (HSL), Red, Green, Blue (RGB), and Cyan Magenta Yellow Key (CMYK) and preview the color before they select their color by clicking the X in the upper left hand corner (upper right hand corner for PC users). Users can click the Help button to receive instructions on using the collaborative whiteboard. Draw Mode, Stroke Size, and Colors are maintained across Whiteboard changes. Each user’s settings are stored locally and therefore completely independent of other users’ settings.
7. Logoffs	WhiteboardServer handleRequest(final String input, final Integer threadNum)	**I/O Space Partitioning** Logoffs - One user with other users working on the same whiteboard, all users on the server log off

	WhiteboardServer handleOutputs(final Socket socket, final Integer threadNum) WhiteboardServer getExistingWhiteboardsAll() WhiteboardServer removeDisconnectedUser(String client, final int threadNum) WhiteboardServer getSameUsersWhiteboard()	**Expected Behavior** Users can logoff the server by clicking the X in the upper left hand corner of the GUI (upper right hand corner for PC users). This logoff will disconnect the user from the server and the Whiteboard they were previously working on. This will also close the GUI on the user's computer. All users working on the same Whiteboard as the user that just logged off should see that the User is no longer working on the Whiteboard. Additionally, the username is now no longer taken and can be reused by new users joining the server. The server should output internal messages indicating that the threads associated with the client are no longer running. When all users have logged off of the server, all states of the whiteboards should be preserved on the server.