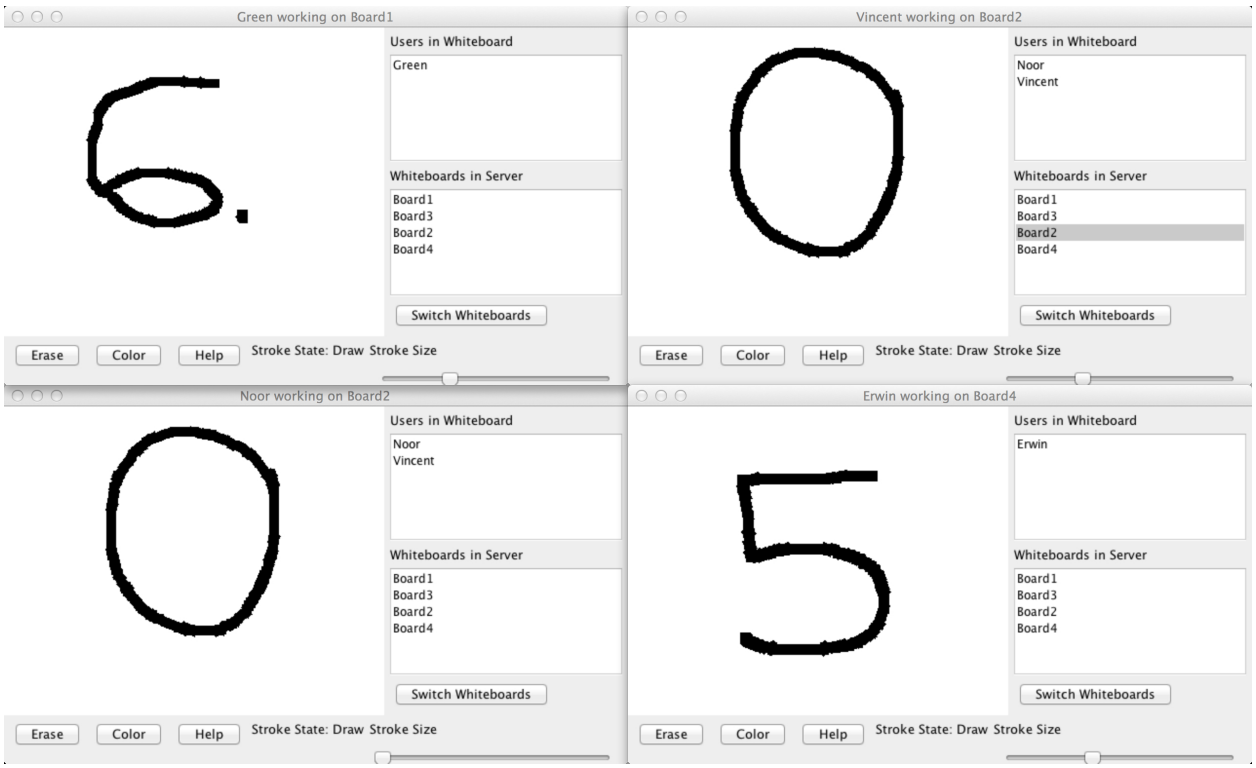
Design Document

Noor Eddin Amer
Erwin Hilton
Vincent Kee



Changes from Milestone

In the milestone design, we focused mainly on developing a fully functional text based message passing protocol and getting the server up and running over telnet. At the time we were working with only two classes, the WhiteboardServer and the WhiteboardClient. The server class was almost fully functional. At the time, the client was also fully functional in terms of connecting to the server and implementing our message passing protocol. The client also included the Canvas model and the GUI view of a Whiteboard in it. The GUI didn't support full functionality but allowed basic drawing and erasing on the canvas.

Upon reviewing our milestone design, we realized that having the view, model, and controller within WhiteboardClient was not modular and not ready for change. The code was starting to become "smelly" and hard to understand. We therefore decided to separate the client code into several different classes. We used the software design principle, separation of concerns, to make adding new features and the debugging process easier. After that, we continued with our implementation until we were able to support minimal functionality.

Our protocol did not change much from the milestone design other than adding in support for users logging off. This allows the Whiteboard server to terminate the threads that maintained the message passing between the now disconnected user and the server.

On the Whiteboard server, we also cut down on the number of threads running simultaneously as well as the number of BlockingQueues. As we tested our design, we found that we did not need a dedicated thread to put incoming messages into a BlockingQueue as well as another thread dedicated to removing items from the queue and executing the appropriate operations. We merged the two threads' tasks into the responsibilities of one thread and eliminated the use of the additional BlockingQueue. On the Whiteboard Client, we did the exact same thing (merging the tasks of two threads into one with handling input from the server).

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 on the server 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 in different colors and with different stroke sizes). 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 and drawing canvas.

A Whiteboard Server is a server that does all the message passing necessary for the users to collaborate in real time using the Whiteboard Clients 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) **main(String[] args)**: Starts a Whiteboard Client using the given arguments. **runWhiteboardClient(final String ipAddress, final int port, final int clientWidth, final int clientHeight):** Runs the Whiteboard Client. **createGUI():** Creates the Whiteboard GUI. **connectToServer():** Connects the client to the server socket and starts the threads to handle inputs and outputs between the server and the client. **handleServerResponse(final Socket socket):** Listens to the socket for messages and passes inputs to the handleRequest. **handleResponse(String input):** Parses input from the server, performing appropriate operations. **handleOutputs(final Socket socket):** Polls the outputCommandQueue and writes items as text messages to the client's socket.
WhiteBoardServer	**C:** public WhiteboardServer(int port) **main(String[] args)**: Starts a Whiteboard Server using the given arguments. **runWhiteboardServer(final int port)**: Starts a WhiteboardServer running on the specified 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 to be handled. **handleRequest(final String input, final Integer threadNum):** Parses client input, performing appropriate operations. **getExistingWhiteboardsAll():** Sends the names of the current Whiteboards on the server to all clients. **getExistingWhiteboardsOne(final int threadNum):** Sends the names of the current Whiteboards on the server to one specified client. **getSameUsersWhiteboard():** Sends out to all clients the names of the other clients working on the same Whiteboard. **removeDisconnectedUser(String client, final int threadNum):** Disconnects the specified client from the server and updates all collaborators with the names of the active clients working on the same Whiteboard. **createBoards()**: Creates 3 Boards for the Server to start with.

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 two threads running per client: one thread to listen for incoming text 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, 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 :== (NEWUSER | ADDBOARD | SELECTBOARD | DRAW | ERASE | DISCONNECT) NEWLINE
NEWUSER :== “new” SPACE “username” SPACE USERNAME
ADDBOARD :== “addBoard” SPACE BOARDNAME
SELECTBOARD :== USERNAME 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
DISCONNECT :== “Disconnect” SPACE USERNAME
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:

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 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. Send all existing Whiteboard names to all users using the following protocol. 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.
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. Send all existing Whiteboard names (including the Whiteboard just added) to all users using the following protocol. 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.

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. Send the names of all collaborators

on the Whiteboard to all collaborators using the following protocol. For each username, append it to a message starting with “sameClient”. Add these messages to each client’s BlockingQueue in the commandsQueue list. When there are no more username messages, add a message saying “Done sending client names.” to each client’s BlockingQueue in the commandsQueue list.

DRAW Message:

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 of each of the collaborators on the Whiteboard.

ERASE Message:

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 of each of the collaborators on the Whiteboard.

DISCONNECT Message:

Closes the socket with the client and terminates the client’s threads on the server. Remove the client from the clientToWhiteboardMap and the clientToThreadNumMap and updates all remaining collaborators with the names of the current collaborators on the Whiteboard.

Grammar for messages from the server to the user:

Server-to-User Whiteboard Message Protocol

MESSAGE :== (USERNAMETAKEN | SELECTBOARD | BOARDADDED | ONBOARD | SERVERBOARDS | BOARDSDONE | UPDATINGCLIENTS | SAMECLIENT | REMOVECLIENT | CLIENTSDONE) NEWLINE

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

SELECTBOARD :== “Select a whiteboard” | “Whiteboard does not exist. Select a different board or make a board.”

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

ONBOARD :== USERNAME “on board” BOARDNAME

SERVERBOARDS :== “Existing Whiteboards “ BOARDNAME

BOARDSDONE :== “Done sending whiteboard names”

UPDATINGCLIENTS :== “Updating Clients”

SAMECLIENT:== “sameClient ” USERNAME

REMOVECLIENT :== “removeClient ” USERNAME

CLIENTSDONE :== “Done sending whiteboard names”

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

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:

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.

BOARDADDED Message:

Sets the Whiteboard name as the client’s Whiteboard name and updates the title in the WhiteboardGUI. Sends message “clientName selectBoard boardName” to the server.

SELECTBOARD Message:

Creates a popup window to get a Whiteboard name from the user. If the Whiteboard name does not already exist on the server, it adds the message “addBoard desiredWhiteboardName” (where desiredWhiteboardName is the Whiteboard name entered by the client) to the outputCommandsQueue. If the Whiteboard name does already exist on the server, it adds the message “clientName selectBoard desiredWhiteboardName” (where clientName is the name of the client and desiredWhiteboardName is the Whiteboard name entered by the client) to the outputCommandsQueue.

ONBOARD Message:

Sets the client’s Whiteboard name to the name of the Whiteboard.

SERVERBOARDS Message:

If the Whiteboard name does not exist in the client’s list of existing Whiteboards on the server, the Whiteboard name is added to that list.

BOARDSDONE Message:

Updates the Whiteboards in Server list in the Whiteboard GUI SidePanel with the client’s list of existing Whiteboards.

UPDATINGCLIENTS Message:

Clears the Users in Whiteboard list.

SAMECLIENT Message:

If the client name does not exist in the client’s list of collaborators, the client name is added to that list.

REMOVECLIENT Message:

Removes the client name from the client’s list of collaborators and updates the Users in Whiteboard list in the Whiteboard GUI SidePanel.

CLIENTSDONE Message:

Updates the Users in Whiteboard list in the Whiteboard GUI SidePanel with the client’s list of collaborators.

DRAW Message:

Draws the line segment defined in the message.

ERASE Message:

Erases the line segment defined in the message.

Graphical User Interface (GUI)

A Whiteboard GUI is the user interface composed of an interactive drawing canvas (discussed in the next section) along with a ButtonPanel and SidePanel. The ButtonPanel offers users the option of drawing in a variety colors, selecting stroke thickness, and even erasing lines. The SidePanel shows the user information such as the other users working on the same canvas and the other canvases on the server. It also allows the user to switch to different drawing canvases if he or she desires to do so.

Class	**Constructor Signature (C) and Operations**
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 **getButtonPanel()**: Getter method for buttonPanel **getCanvas()**: Getter method for Canvas **getUsername(String message)**: Gets the desired username from the user, checking against the existing usernames. Usernames CANNOT contain spaces or just be an empty string. The “message” string is appended **chooseWhiteboardPopup(String desiredWhiteboardName):** pops up a dialog box that allows the user to input his/her desired username. If username is already taken or invalid (empty string or contains spaces), the popup box pops up again. **chooseNewWhiteboard(String desiredWhiteboardName)**: 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
ButtonPanel	No methods. **Java SWING GUI Components:** private JButton drawButton; private JButton helpButton; private JButton colorButton; private JSlider strokeSize; private JLabel strokeSizeLabel; private final JLabel strokeState;

	Action Listeners: 1) drawButton Action Listener changes the strokeState label (to show whether or not the cursor is in draw or erase mode). 2) helpButton Action Listener opens up a popup box with the helpBox() command from the WhiteboardGUI with the appropriate help message 3) colorButton Action Listener opens up a popup box with a color chooser that allows the client to change the color of the draw method. 5) The strokeSize slider sets the size of the canvas's stroke to the value the slider is pointing at, whenever the slider is not adjusting - set only when stationary.
SidePanel	**updateWhiteboardsList(List<String> whiteboards, String currentWhiteboard):** Updates the whiteboards JList appropriately (when new whiteboard is added). Is called by the WhiteboardClient **updateClientsList(List<String> clients):** Updates the clients list appropriately (when a client disconnects, is added, etc). Is called by the WhiteboardClient **Java SWING GUI Components:** private JScrollPane usersInWhiteboard; private JLabel usersInWhiteboardLabel; private JScrollPane whiteboardsInServer; private JLabel whiteboardsInServerLabel; private DefaultListModel<String> usersInWhiteboardListModel; (for the usersInWhiteboard jlist) private DefaultListModel<String> whiteboardsInServerListModel; (for the whiteboardsInServer jlist) private JList<String> usersInWhiteboardList; private JList<String> whiteboardsInServerList; private JButton selectWhiteboard; **Action Listeners:** **1)** whiteboardsInServerList JList Action Listener reads the selection of the client in the JList and changes the selectedWhiteboard string to the value of the selection (this alone does not switch the selected whiteboard of the client, that is only done after the selectWhiteboard button is clicked) **2)** selectWhiteboardButton Action Listener changes the client's whiteboard to the value of the selection in the whiteboardsInServerList JList using chooseNewWhiteboard(String selectedWhiteboard). When nothing is selected as the client is initialized, the client remains in the current whiteboard when the button is pressed.

Datatype Design

A Canvas is an interactive drawing canvas composed of 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 WhiteboardServer. The user can draw and erase pixels on the Canvas by clicking and dragging their mouse cursor across the interactive surface. The ButtonPanel and SidePanel on the WhiteboardGUI give the user additional options to interact with the Canvas.

Class	**Constructor Signature (C) and Operations**
Canvas	**C:** public Canvas(int canvasWidth, int canvasHeight, BlockingQueue<String> queue) **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. **setWhiteboardName(String name):** sets the whiteboard name **getWhiteboardName():** Getter for the whiteboard **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 **getStrokeState():** returns the stroke size **getTcc():** gets the colorchooser tcc for the board **setServerColor(int red, int blue, int green):** sets the color of the server color chooser, taking in int red, int green, int blue values (rgb). The server color chooser is used to display the color information of a user's drawing to the rest of the user's on the server/whiteboard. **setClientColor(int red, int blue, int green)**: sets the color of the client color chooser, int red, int green, int blue values (rgb)

	getServerColor(): gets the color of the server color chooser, in an array of { int red, int green, int blue} values (rgb). **getClientColor()**: gets the color of the client color chooser, in an array of {int red, int green, int blue} values (rgb) **getCanvasWidth():** gets the width of the canvas **getCanvasHeight():** gets the height of the canvas **getDrawMode():** gets the drawMode of the canvas **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, and 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

Separation of Concerns

In this section, we discuss our implementation of the software design principle, the separation of concerns, in our design. Our code is separated into different modules, each concerned with a different task. Canvas represents our fundamental model that supports drawing, erasing and updating the Image drawingBuffer according to the clients' input and server commands. WhiteboardGUI is in charge of running the graphical user interface, handling the input from the user using different action listeners, and sending text commands to the client's output queue. The WhiteboardClient is in charge of communicating with the Whiteboard Server and calling the appropriate methods in WhiteboardGUI according to command messages received from the server. WhiteboardServer is in charge of supporting collaboration between multiple active clients on multiple whiteboards simultaneously.

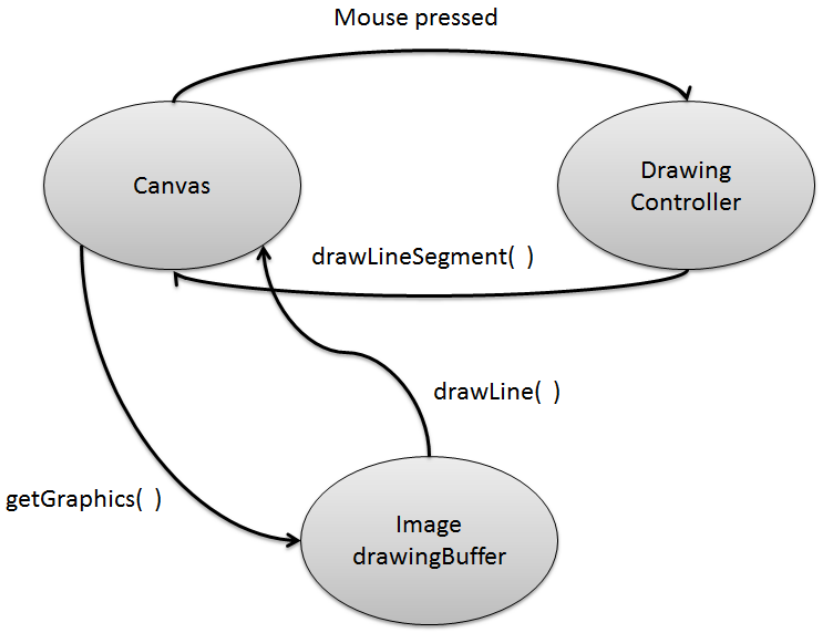
In this project, we based our design on a pattern similar to the model-view-controller (MVC) pattern. We split our design into a model (Canvas), a view (WhiteboardGUI), and a controller (Whiteboard Client). The design didn't strictly follow the MVC pattern as our model and GUI were interconnected. We now discuss the different parts of our design, the model and the view, (the controller aspect was discussed earlier with the Client Server Model) and how they communicate with each other.

The Model

Our model is the Canvas class. Canvas represents an interactive drawing surface. It supports drawing and erasing in different stroke sizes and colors, with methods for changing the drawing parameters. Canvas therefore represents the fundamental model for a whiteboard. Different views can be implemented to represent this model.

It is worth noting though that the Canvas class itself is a JPanel. Inherently, the data of the Canvas is based on direct input from the user (which it also displays) and therefore it does not strictly fit being the model in a Model-View-Controller pattern, although it acts in many ways as one.

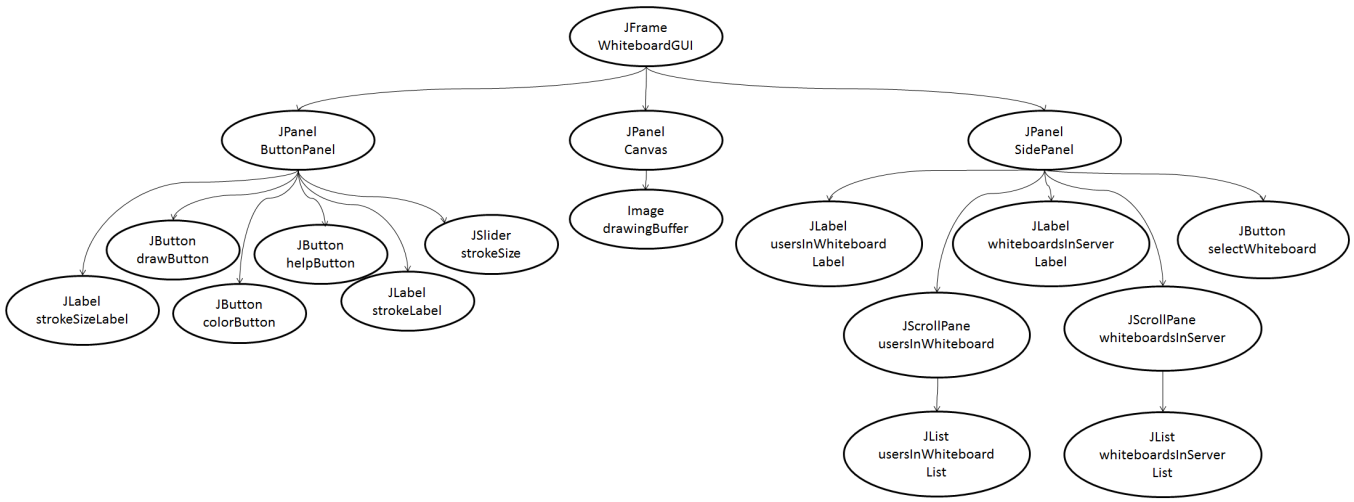
The MVC pattern is seen in our Canvas implementation. The Image, drawingBuffer, represents the model, Canvas represents the view, and DrawingController represents the controller. Canvas extends JPanel and displays the model state in the GUI. The DrawingController class uses the Listener pattern to listen for user input (by looking at actions taken on the GUI such as mouse clicking and dragging) and calling the appropriate methods in Canvas. Those methods change the Image (drawingBuffer), which represents the model in that case. The figure below illustrates this pattern when the method drawLineSegment is called.



The View (WhiteboardGUI)

In our attempt to implement separation of concerns, we decided to completely separate the View from the Controller. We created a new class, WhiteboardGUI, that extends JFrame and includes all the components of the user interface. WhiteboardGUI includes instances of three classes within it, ButtonPanel, SidePanel and Canvas, all three of which extend JPanel.

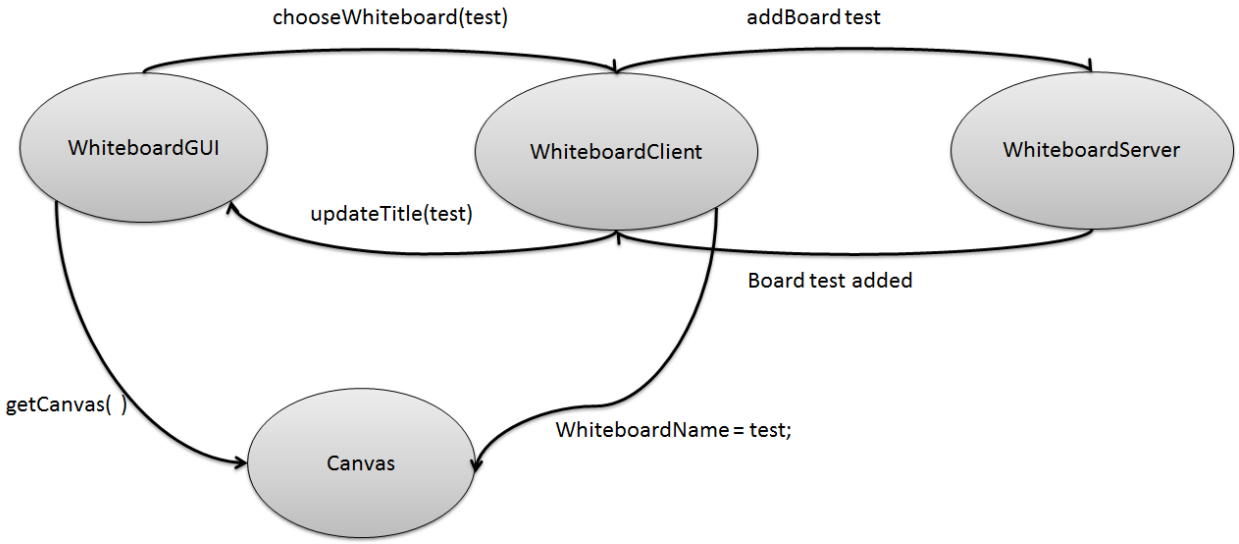
We also tried to make the user interface more modular by implementing the Composition pattern within the classes comprising the user interface. The ButtonPanel is responsible for displaying the toolkit that the client can use in editing Canvas. The SidePanel is responsible for providing information from the server including the users working on the same whiteboard and the whiteboards available for the user to switch to. As explained in the model section, although Canvas represents our fundamental model, it also includes a user interface that enables the user to draw on its image, drawingBuffer. Below is a view tree of our graphical representation of a Whiteboard.



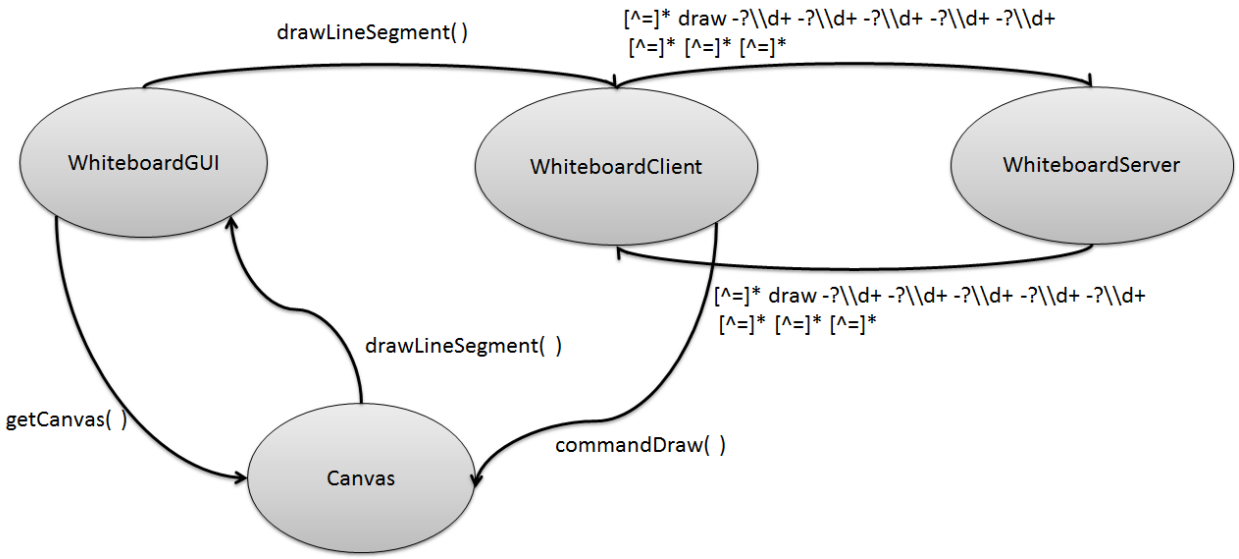
Putting It All Together

Delegating an instance of Canvas in WhiteboardGUI made it easier for us to support communication between the WhiteboardClient and the GUI. We included methods in the WhiteboardGUI, such as getUsername and chooseNewWhiteboard, which give the client access to all relevant information for communication with the server. This created a pattern of communication between the model, the view, and the controller that supports the intended functionality in a modular fashion.

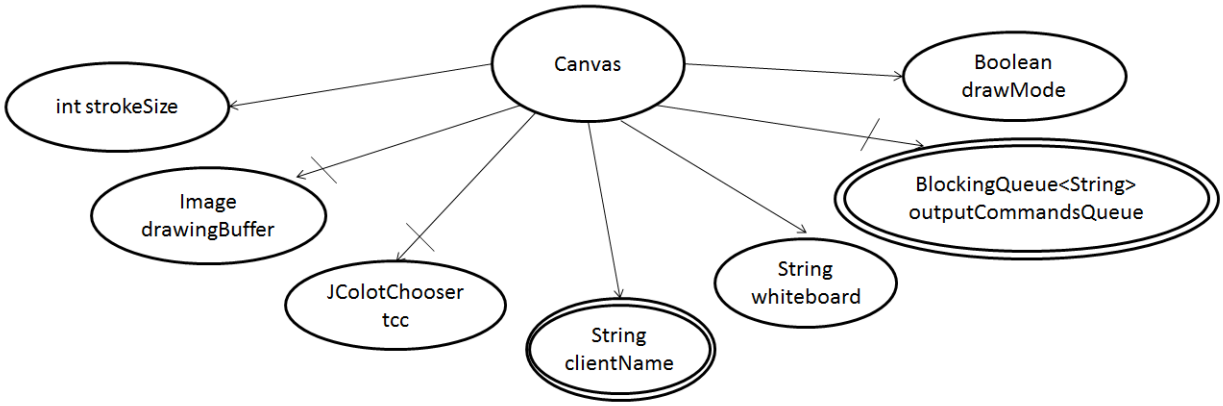
The figure below shows how this pattern of communication works. When the chooseWhiteboardPopup method is invoked in the WhiteboardGUI, the String “addBoard test” is added to the client’s output commands queue. The server responds to this command with the String “Board test added”. This command is sent to the client, which calls WhiteboardGUI’s updateTitle method to update the view’s title. It also accesses the Canvas through WhiteboardGUI’s getCanvas method and updates the whiteboardName field in Canvas.



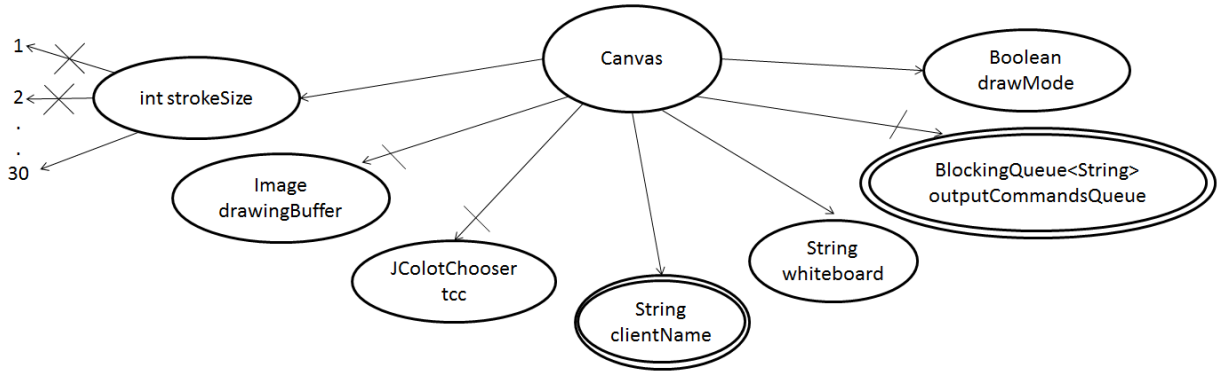
This pattern is seen over and over again in our design. When the user does an action in the GUI, an action listener is invoked, putting a command in the WhiteboardClient's outputCommandsQueue. The WhiteboardClient then sends the command to the Whiteboard Server, which responds to that command and sends the appropriate response back to WhiteboardClient (and all clients depending on the command). Finally, the WhiteboardClient invokes methods in WhiteboardGUI to update its state and then accesses the Canvas through WhiteboardGUI and updates its state to be identical to that of all the board's collaborators. This can be seen the figure below.



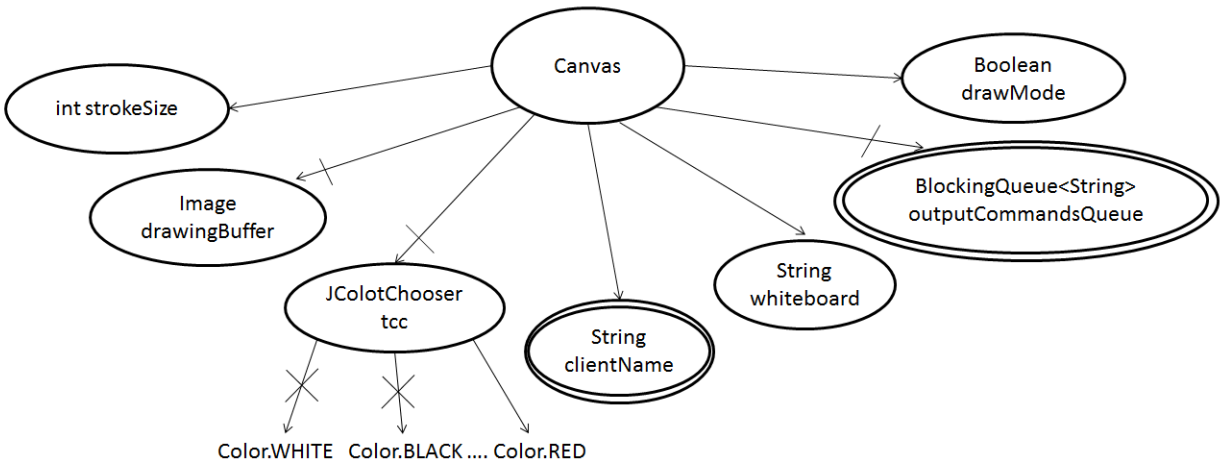
Snapshot Diagram of a Canvas in Action



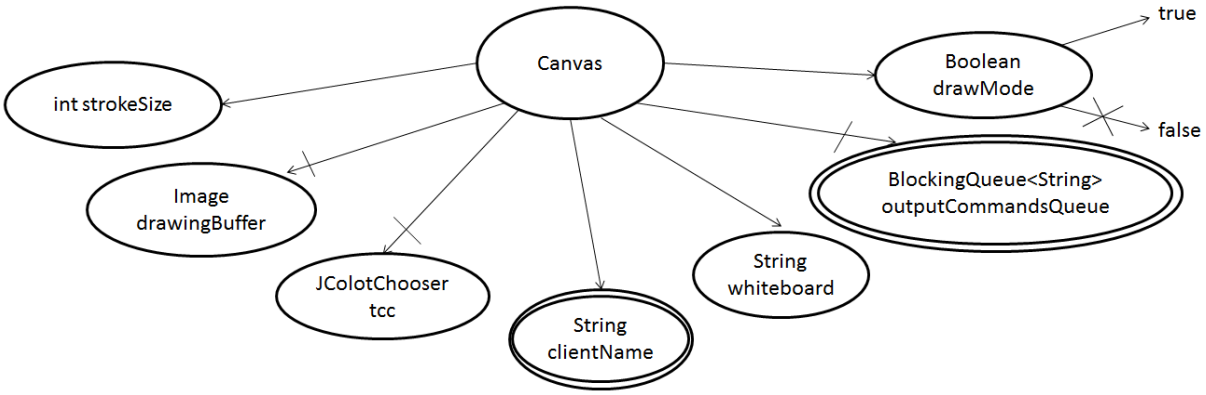
Changing Strokesize



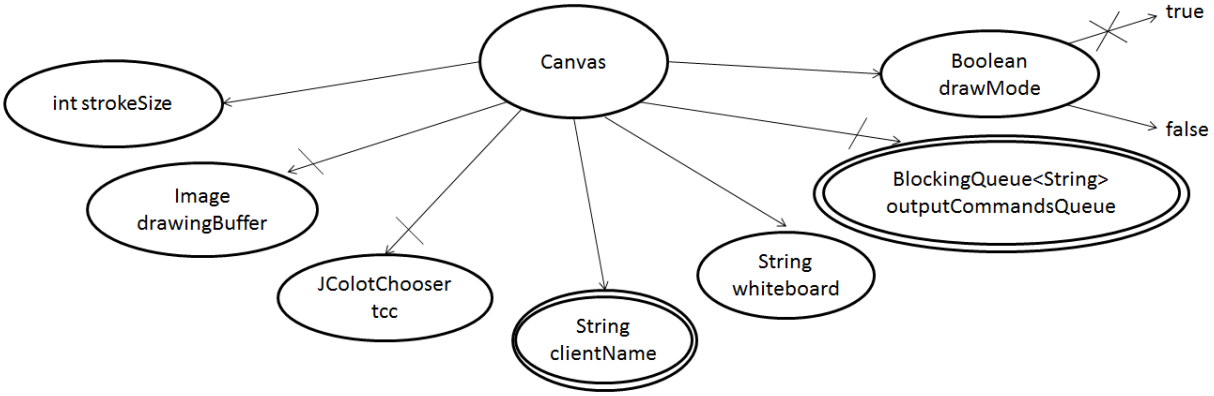
Changing Colors



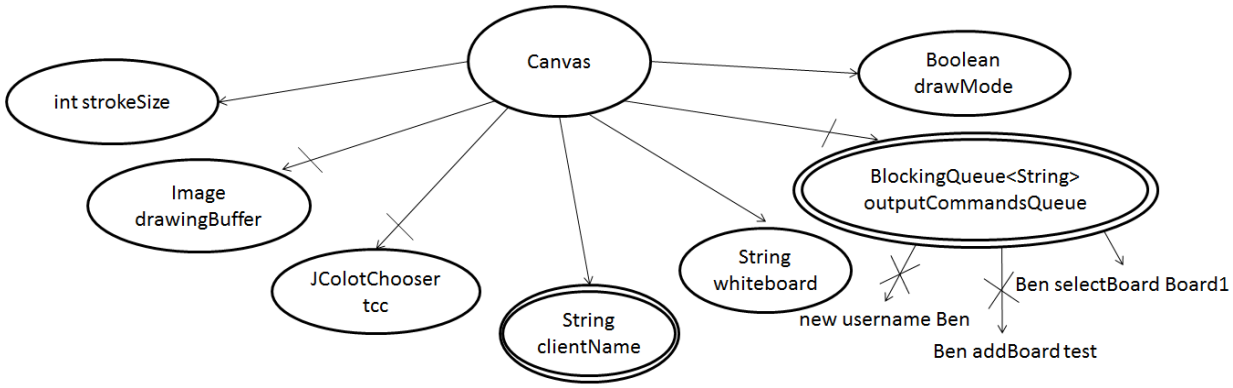
Switching Erasing to Drawing (drawMode to true)



Switching Drawing to Erasing (drawMode to false)

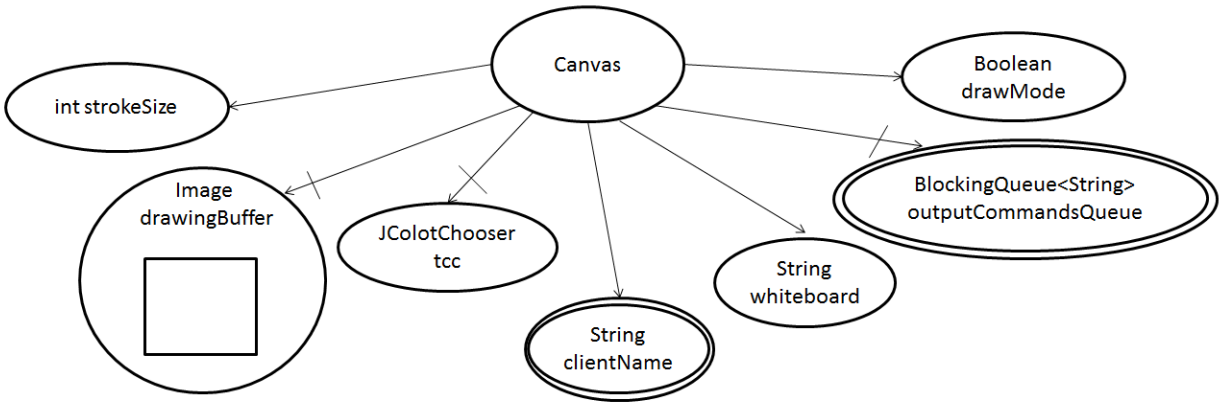


Offering Messages to Queue

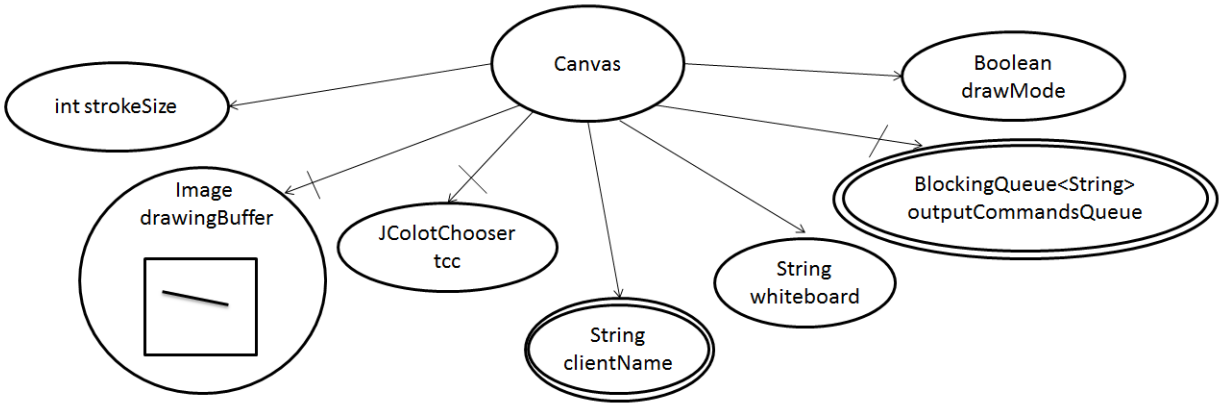


Drawing a line to Image

Before

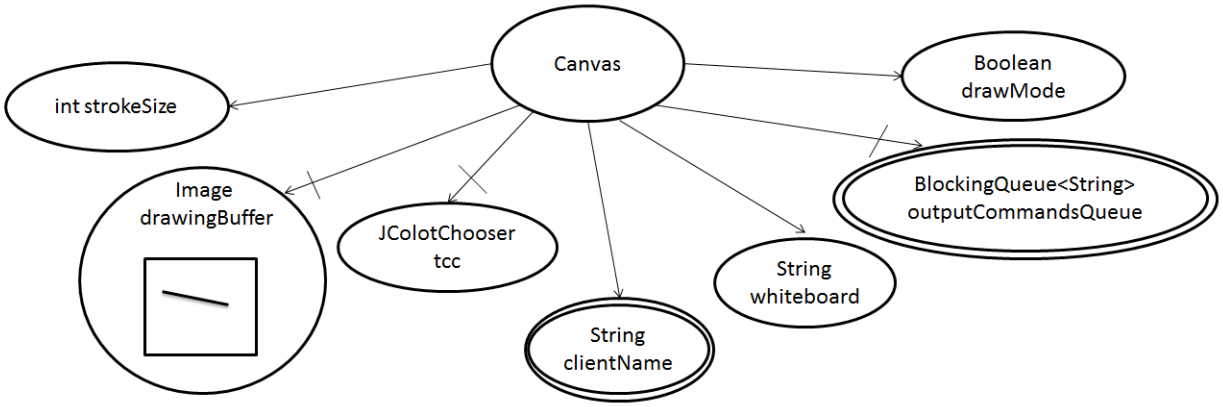


After

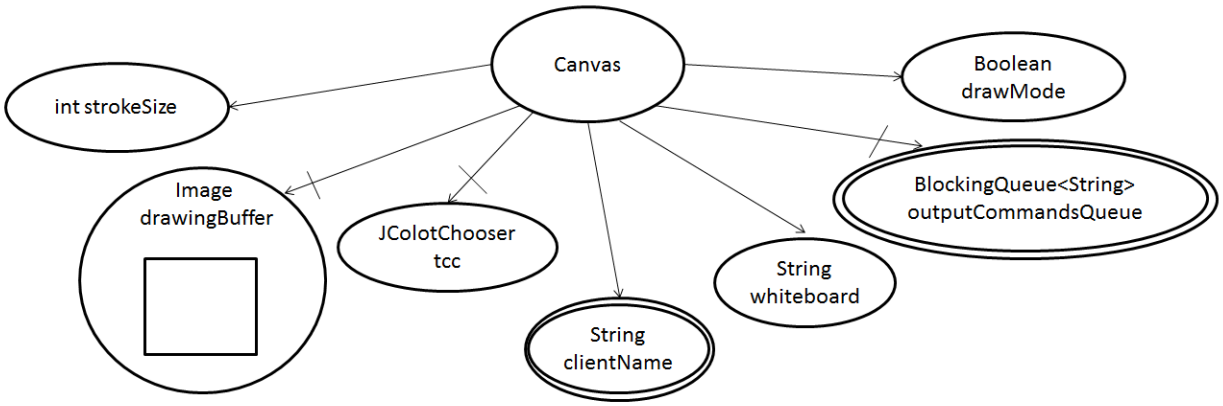


Erasing a drawn line from Image

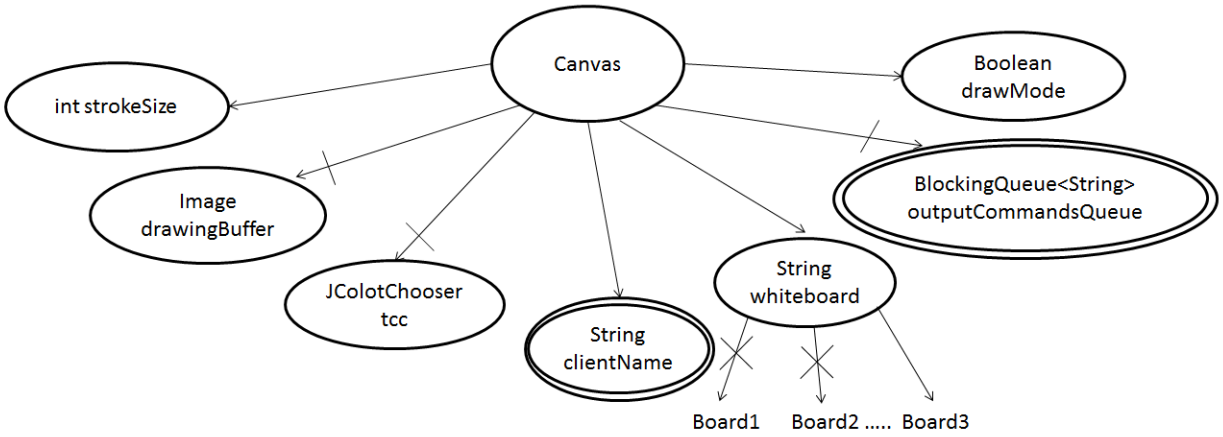
Before



After



Changing Whiteboards



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. Only one thread adds commands to each BlockingQueue (using offer) and similarly only one thread removes commands from each BlockingQueue (using take) so there are no risks of race conditions. 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 and update 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. The outputCommandsQueue that stores the commands to be sent to the Whiteboard server is a Blocked Queue and only one thread adds items to it (using offer) and similarly only one thread removes commands from it (using take). Therefore, there will be no race conditions as items are added and removed from it. 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 the Server, Client, GUI, and Canvas 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

In testing our Whiteboard canvas, we will use both JUnit and manual tests.

The Whiteboard Canvas checkRep() method is run every time the canvas datatype is mutated, 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)**
Setting Color of the Client's Drawn Lines	setClientColor(int red, int blue, int green) getClientColor()	**I/O Space Partitioning:** Input - (R,G,B) values of all 0, all 255, and different combinations in between. **Expected Behavior:** The client's color chooser color is expected to be set to the specified RGB values.
Setting Color of the Collaborators Drawn Lines	setServerColor(int red, int blue, int green) getServerColor()	**I/O Space Partitioning:** Input - (R,G,B) values of all 0, all 255, and different combinations in between. **Expected Behavior:** The server's color chooser color is expected to be set to the specified RGB values.
Setting	setWhiteboardName()	**I/O Space Partitioning:**

Whiteboard Name	getWhiteboardName()	Input - Strings with lowercase, uppercase letters, numbers, and special characters. Spaces/empty strings are taken care of by the client/server (so are not tested for). **Expected Behavior:** The whiteboard name of the canvas is set to the client's chosen whiteboard
Setting the strokeState (size) of the draw/erase methods	setStrokeState() getStrokeState()	**I/O Space Partitioning** StrokeSize values - zero, small integers, large (MAX_INT), and negative integers (negative integers automatically set to zero) **Expected Behavior** The strokeState (size of the draw/erase stroke is changed corresponding to the appropriate value)

Manual testing for full functionality of the Whiteboard canvas will be done in the end to end tests as discussed below in section 3.

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 Auto-creating 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	**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

	getExistingWhiteboards One(threadNum)	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, Canvas, GUI (Whiteboard GUI, ButtonPanel, SidePanel)

We used manual end to end tests to test our server, client, and GUI's functionality and performance as various stages in the control flow 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)	**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).

	WhiteboardClient connectToServer()	
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) WhiteboardClient handleServerResponse(String input) WhiteboardClient createGUI() WhiteboardGUI getUsername(String message)	**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. Computers - Connecting with one computer at a time, multiple computers simultaneously **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.

		Note that no username conflicts with simultaneous user submissions as the username is granted to the client that connects to the server first.
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() WhiteboardClient createWhiteboard() WhiteboardGUI chooseWhiteboardPopup()	**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 o 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. Computers - Connecting with one computer at a time, multiple computers simultaneously **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 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. Note that only one Whiteboard can be created per name as simultaneous new Whiteboard submissions will create a new Whiteboard with the command that is received first and subsequent new Whiteboard commands will simply select the board.
	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 addDrawingController() Canvas checkRep()	**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.
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	**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

	newStroke) Canvas setStrokeState (int value) Canvas getTcc() WhiteboardGUI helpBox()	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. Note that 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) WhiteboardServer handleOutputs(final Socket socket, final Integer threadNum) WhiteboardServer getExistingWhiteboards All() WhiteboardServer removeDisconnectedUs er(String client, final int threadNum) WhiteboardServer getSameUsersWhitebo ard()	**I/O Space Partitioning** Logoffs - One user with other users working on the same whiteboard, all users on the server log off **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. When a new user joins a Whiteboard, they should see all the history displaying on their canvas.