Design Document: Milestone

Status of Project

We have the server class (WhiteboardServer.java) fully functional and implemented as specified in the document except that it does not use a BlockingQueue to store inputs from clients but rather handles them directly. Instead of having a thread that constantly checks for inputs on the socket and places them all into a BlockingQueue for inputs, it handles them directly. The thread that constantly polls that BlockingQueue and handles them thus has not been implemented. This is because our implementation of this (WhiteboardServerTripleThreadNonWorking.java) seems to nondeterministically output to the wrong client. We were unable to resolve this issue at the LA Office Hours on Monday.
We have however tested our Whiteboard Server class (without the input Blocking Queue) using telnet and it has been fully functional with multiple clients.
We have almost finished implementing the client class. We will be separating the client from the whiteboard to make the code more modular.

General Control Flow

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 immediately has a popup window requesting a username. This username is sent to the server. If it is not unique, the same window will query for a different username and display the taken usernames.
2) The user is prompted to select a whiteboard from the existing whiteboards or create a new whiteboard. The users working on each whiteboard are displayed next to each whiteboard.
3) Now the Whiteboard user interface loads with the state of the Whiteboard as stored on the server.
4) The user can make any changes they want to the drawing canvas. 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.
   At any point the user can now
   a) Send messages to the other users working on the same Whiteboard.
   b) 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, with all past changes present.
   c) 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.

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.

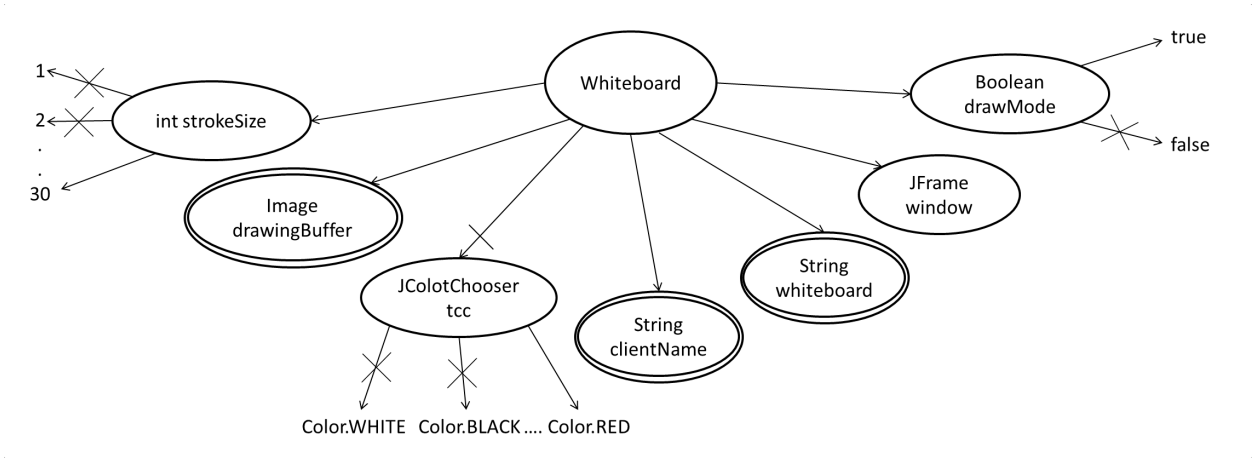
Class	**Constructor Signature (C) and Operations**
WhiteboardClient	**C:** public WhiteboardClient(Whiteboard whiteboard) **connectToServer():** Connects the client to the server socket and starts the threads to handle inputs and outputs between the server and the client. **handleServerResponse():** 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.

Class	**Constructor Signature (C) and Operations**
Whiteboard	**C:** public Whiteboard(int width, int height) **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. **addDrawingController()**: Adds the mouse listener that supports the user's freehand drawing. **makeCanvas()**: Sets up the User Interface **fillWithWhite()**: Makes the drawing buffer entirely white. **setStrokeState(int value)**: Sets the size of the stroke **helpBox():** Pops up a window with help information **colorChooser():** Pops up a window that allows the user to choose color to draw. **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

Snapshot Diagram of a Whiteboard in Action



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 private and final 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 ensures that the correct messages are 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 Concurrent HashMaps, which prevent race conditions as different threads access the information to properly distribute information to the correct clients. 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 so there is no representation exposure. Like in the Whiteboard Server, the inputCommandsQueue and outputCommandsQueue are 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 Whiteboard

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 in our datatype definition are confined to one thread (thread confinement). As a result, no locks were used on the Whiteboard. Therefore, there is no risk of deadlock on the Whiteboard. Therefore, the Whiteboard 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 user interface 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 for race conditions that would otherwise hard to reproduce manually. 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. Whiteboard

In testing our Whiteboard canvas, we will use both JUnit and manual tests. The JUnits tests will test that the representation invariant is maintained while all the methods are called. Our checkRep() method will be run in each test, asserting that none of the fields in the Whiteboard class are null. For the draw and erase methods, we will check whether the canvas has been drawn on using color checks (checking Color.BLACK, Color.WHITE,etc) on the appropriate pixels to make sure the methods are modifying the canvas correctly.

To test the proper functionality of the Whiteboard canvas, we manually will run the GUI and visually verify the correct functionality of each method.

2. Protocol

We will test our protocol and server manually using telnet and using JUnit tests. For single client protocol testing, we will use JUnit tests as it is relatively easy to implement. This is possible because all message passing commands are strings, which allow for us to quickly verify that the server sent the appropriate response. However, with various multiple client tests, we will use manual tests as it will be significantly more complicated and time intensive to create JUnit tests for testing correct server behavior. With telnet, we will execute all feasible text commands combinations on telnet and see if we get the expected responses. Our testing strategy will primarily focus on verifying that the correct clients received the responses intended for them and that the responses were valid.

3. GUI

The user interface will also be tested manually. We will start by testing the functionality of each actionListener and method in the GUI. Each method will have its own input and output space that will be tested accordingly. For example, for the erase method, we will test that erasing takes place when the erase button is pressed and for the username entry window, we will verify that the username is properly set.

4. End to End testing

We will use the GUI to manually conduct the end to end testing and testing the full functionality of the client as well as the performance of the server.

We will start by testing for full functionality with one client. For the coverage in testing the fields, we will try different possible string inputs for the usernames and whiteboards, strings of various length and types of characters, and the empty string.

After testing for functionality with one client we will test the functionality with multiple clients. We will test that the basic requirements are fulfilled like the server supporting multiple whiteboards being modified simultaneously, no duplicate client and whiteboard names being allowed, clients being able to switch between whiteboards, clients working with whiteboards at their current state, and whiteboard state persisting over client disconnects and reconnects. We will test that the performance is not affected when multiple users are performing actions simultaneously and keep a list of string logs on the client and server so that we can ensure that the client commands are executed in the order that they are received at the server and client ends. All three of us will try connecting to the same whiteboard and test responses to potentially troublesome behaviors like erasing the same section from two computers and drawing in that position from another computer simultaneously. We will also see whether each of us is able to see the usernames of the others as well as the other whiteboard names.

Finally, we will test the additional features that we implement.