

Harihar Subramanyam
Zeke Schmois
Robert Perez

6.005 Project 2 – Design Document

ADTs:

There are four main classes that constitute the ADTs that manage the whiteboards, users, drawn lines, and interactions between them. They are:

Line: This is an immutable datatype which represents a line drawn on the board. It consists of a stroke thickness, a color (specified by the alpha, red, green, and blue components), and two (x,y) pairs marking the endpoints of the line. The class also includes a toString() method which returns a string that can be used to send line data between client and server.

User: This datatype encapsulates the two main aspects of a user – name and ID. The name is a string which reflects the “human-readable name” of the user (ex. Harihar, Zeke, Robert). The ID is an integer which is unique to every user (this uniqueness is ensured by the LobbyModel, which is responsible for creating users). The user’s ID is immutable, but the name can be changed if needed.

Whiteboard: This datatype represents a whiteboard and its contents. Like the User class, it has a name field and an ID field. The name is the “human-readable name” of the board (ex. 6.005 planning, Lecture notes, Pictionary). The ID is an integer which is unique to every board (the uniqueness is ensured by the LobbyModel, which is responsible for creating whiteboards). Unlike a User object, however, a Whiteboard object also has a List of Line objects (i.e. List<Line>) which represent all the lines that have been drawn on the board. The last Line in the list is the most recent one drawn. It provides a method for adding lines and another method for clearing the board. Note that an individual line cannot be deleted. This was done to simplify the process of managing lines.

LobbyModel: This is the most important ADT because it combines the Line, User, and Whiteboard classes to create a representation of a “lobby” which contains a number of whiteboards, each with some users collaborating on them. The class includes two AtomicInteger objects, which are incremented each time a user or board is created (the integer then becomes the ID of the board/user). The class also includes three Maps:

Map<Integer, User> userForID

(the key is an ID of a user, and the value is the user with the given ID),

Map<Integer, Whiteboard> boardForID

(the key is an ID of a board, and the value is the board with the given ID)

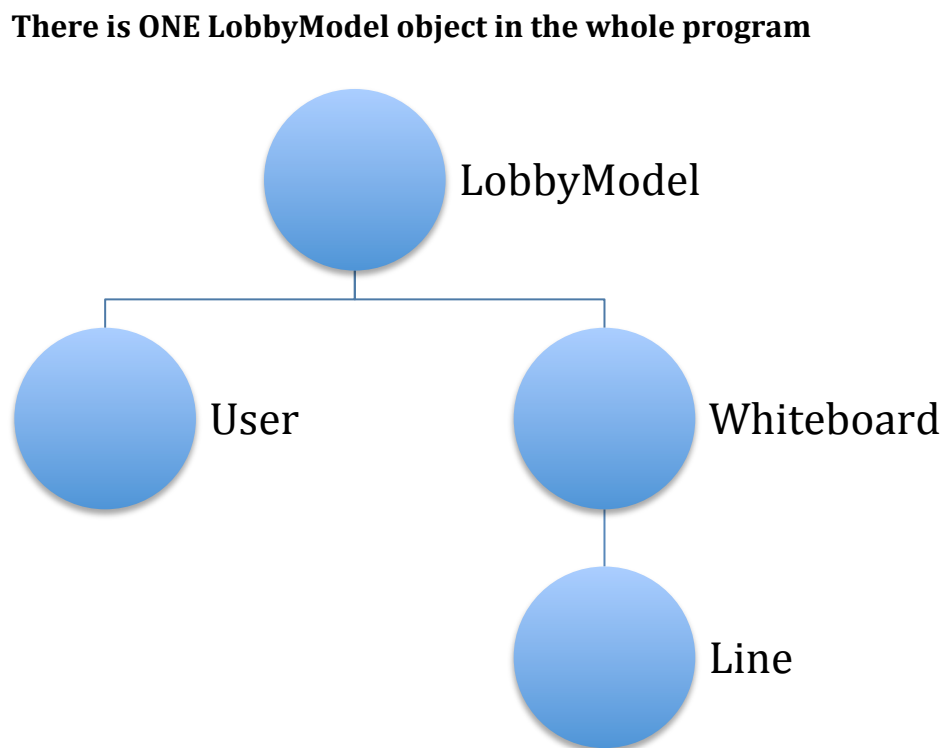
`Map<Integer, Set<Integer>> userIDsForBoardID`

(the key is an ID of a board, and the value is a set containing the user IDs of all the users in the board with the given board ID).

The LobbyModel is the key ADT, because all attempts to manipulate the boards and users must go through the LobbyModel.

For instance, to create a user, we must call `myLobbyModel.addUser(aUserName)`, to join a board we must call `myLobbyModel.userJoinBoard(userID, boardID)`, and to add a line to a board, we must call `myLobbyModel.addLine(line)`.

The LobbyModel contains all the methods needed to mutate and retrieve information about whiteboards, users, the relationships between them. The hierarchy of the classes is shown below.



Server Side:

There are three classes which handle the connection and handling of client messages.

MessageHandler: This class provides a single method `handleMessage(String input, UserThread thread, LobbyModel lobbyModel)`, which takes the user's input, thread, and the lobbyModel. The method processes the input, calls the appropriate methods on the lobbyModel, and outputs a response to the user thread. Along with the LobbyModel class, the MessageHandler class is the main workhorse of the server.

UserThread: This class is responsible for handling the connection of a single user. It serves only one purpose – to read the user's input and pass it to the MessageHandler.

WhiteboardServer: This class is responsible for accepting user who connect to the server and creating a UserThread for that user. It also instantiates the SINGLE LobbyModel object which is used throughout the program.

Protocol:

These are the following requests and responses supported by the server.

Req: `get_board_ids`
Resp: `board_ids [id1] [id2] [id3]`

Req: `[newUserName]`
Resp (to all users in board): `users_for_board [boardID] [userName1] [userName2]...`
Resp (to users who made request): `done`

Req: `create_board [boardName]`
Resp (to all other users): `board_ids [id1] [id2] [id3]`
Resp (to user who made request): `done`

Req: `get_current_board_id`
Resp: `current_board_id [boardID]`

Req: `get_users_for_board_id [boardID]`
Resp: `users_for_board [boardID] [userName1] [userName2]...`

Req: `join_board_id [boardID]`
Resp (to all users in board): `users_for_board [boardID] [userName1] [userName2]...`

Resp (to user who made request): board_lines [x1] [y1] [x2] [y2] [strokeThickness] [r] [g] [b] [a] [x1] [y1] [x2] [y2] [strokeThickness] [r] [g] [b] [a] [x1] [y1] [x2] [y2] [strokeThickness] [r] [g] [b] [a] [x1] [y1] [x2] [y2] [strokeThickness] [r] [g] [b] [a]...

Req: logout

Resp (to all users in board): users_for_board [boardID] [userName1] [userName2]...

Resp (to user who made request): logged_out

Req: get_users_in_my_board

Resp: users_for_board [boardID] [userName1] [userName2]...

(if not in a board): failed

Req: leave_board

Resp (to all users in board): users_for_board [boardID] [userName1] [userName2]...

Resp (to user who made request): done

Req: req_draw [x1] [y1] [x2] [y2] [strokeThickness] [r] [g] [b] [a]

Resp (to all users in board including user who made request): draw [x1] [y1] [x2] [y2] [strokeThickness] [r] [g] [b] [a]

(if not in a board): failed

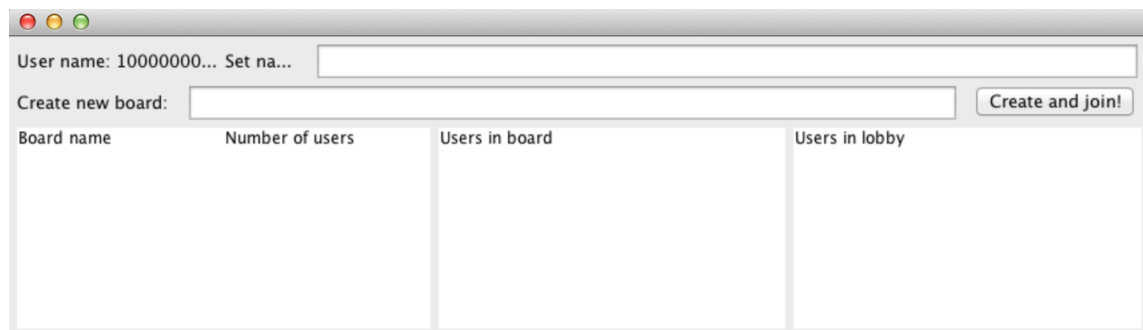
Req: req_clear_board

Resp (to all users in board including user who made request): clear_board

Client Side:

There are two View-Controller pairs on the client side.

LobbyGUI: This represents the lobby (i.e. the state where a user has not joined a board yet and is looking for one). The user interface currently looks like this:



User name: 10000000... Set na...

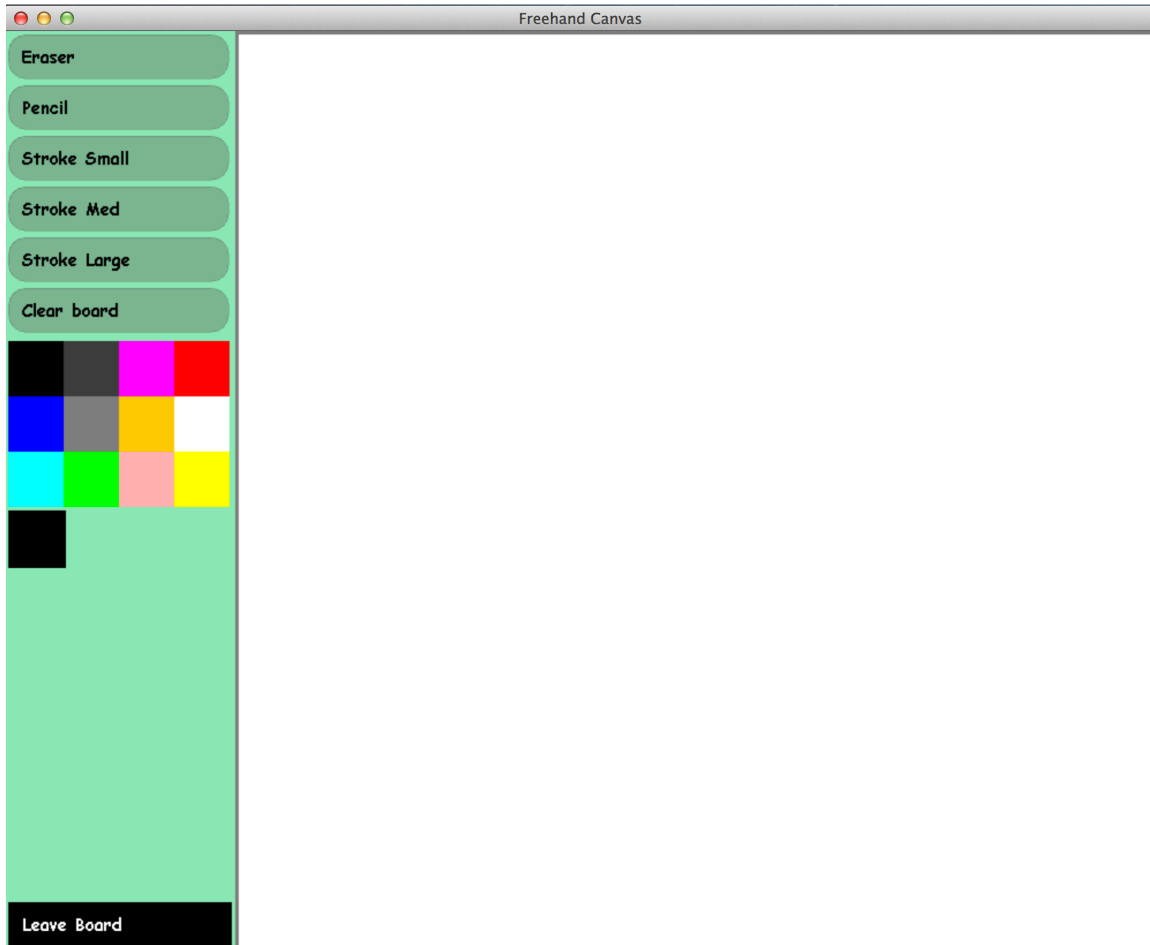
Create new board:

Board name	Number of users	Users in board	Users in lobby

The controller makes the connection to the server and launches a separate thread to read the input stream from the server and update the UI whenever server responses come in. The Swing thread provides ActionListener objects which take

user input (ex. enter a username, enter a board name, clicking a button) and create server requests and send them on the output stream to the server.

Canvas: This represents the actual whiteboard that the user draws on. The current user interface is displayed below:



A listener has been created to detect drags. Dragging the mouse creates lines, and these are turned into Line objects which are sent to the server. The server updates its model and then broadcasts the Line object to all users in the board. When they receive the response (as in LobbyGUI, we create a separate thread to monitor the input stream from the server and update the UI when responses arrive), they draw the line on the screen.

Concurrency Strategy:

We consider the server side first. Recall that the server side behavior is encapsulated into three classes. The WhiteboardServer class waits for users to connect and then gives them a UserThread. The UserThread class is responsible for reading user input and passing it off to the MessageHandler. The MessageHandler then uses the LobbyModel to process that input and sends the response using the

output stream of the UserThreads. To ensure that race conditions do not jeopardize the program, we put a lock on the MessageHandler's handleMessage(String message) method. This way, multiple requests do not interleave and corrupt the LobbyModel. Secondly, we put a lock on the UserThread's output(String message) method. This way, a UserThread can only output a message after it's finished outputting its previous message. We also use immutability wherever possible (ex. Line objects are immutable, all IDs are immutable, etc.). We also use confinement and access data structures with a "dedicated thread" whenever possible (ex. the thread in which the MessageHandler is located is the only thread that can change the LobbyModel). Finally, on the client side, we have one dedicated thread for receiving messages and updating the UI – other threads cannot do that.

Testing Strategy:

We aim to do as much testing as possible. For instance, we will write tests for the MessageHandler which test every message in the protocol and ensure that they return the correct responses. We will also test every method of each of our ADTs – Line, User, Whiteboard, and LobbyModel. Finally, we will perform manual tests of the UI, and document them thoroughly.