Project 2 | Design Milestone

6.005: Software Construction (Fall 2013)

Team Members

- Dongwei Jiang (mitjdw)
- Nicholas Matuzita Mizoguchi (nickmm)
- Rebekah J. Cha (rcha)

Major Changes

Application flow

Explains how application flow, showing how objects are instantiated, and how threads behaves in the solution's environment.

Client-Side refactored

More ready for change. Enabled an easy integration with the android app. The changes were:

1. ApplicationClient now named ClientApplication.
2. ClientApplication doesn't have the static main method. A Main.java file was created.
3. ClientApplication now has a ClientListener object. The ClientListener interface has callback methods that handles each type of message received.

Created CWPMessage.java class

A class that treats messages of our custom protocol (Collaborative Whiteboard Protocol). With this class is possible to validate messages, and also create messages in the protocol's format.

Supporting names with spaces

Designed the protocol to accept spaces. This was made by setting a token SEPARATOR different then a space. Now the SEPARATOR is set to the character '\0'.

Java Swing GUI better organized

The GUI is divided into different sections of the display. The canvas remains in the middle but there is a MenuWest on the west side, MenuNorth on the North, and MenuEast on the east side of the canvas.
In MenuWest, we changed how the user could set the brush size by adding + and - button which increment and decrement the size by 1 and added a text box for the user to enter in a size. We added a toggle button called fillColor that allows the user to select if the rectangle should have a fill color and also select what color.
In MenuNorth, we added a File-> Save As function that saves a png file of a screenshot if the display into the user selected directory.
In MenuEast, we implemented the list of online users that are looking at the same board, a list of whiteboards to look at, and a chat box. The chat box announces to all of the users when a user

connects or disconnects, and also allows the users to input messages to each other.
For the tools that are used when drawing on the canvas, we have different controllers associated with them that can freedraw, draw lines, or draw rectangles. Depending on which tool the user selects, a message is sent to the server amd calls the respective tool.

Unique ID's for Users

Now users can have same usernames, and still be unique. This solves the problem of duplicated usernames.

New message: updateusers user1 … user *n*

A new message was supported, so we could know who was connected to the current active board. The server now updates the client every time someone connects, or changeboard. All clients that should receive updates of changes now receive it and update their list of online users in their current board.

Thread-safe arguments better explained

Now we have all arguments saying specifically which thread-safe datatypes are being used.

Testing for integration

Integration tests were implemented, so we could test the behavior of the server as a whole, and also the client. Server and Client can be tested separately.

Android App

We developed an Android version of the solution. Since Android also uses Java, all the code from the models were entirely reused. Since the ClientListener is the interface between the client model and the GUI, we only needed to implement the GUI for Android, using the interface to gather all the information. Android has a Android Graphics library, which is very similar to the Java Graphics2D library. We did implement some additional activities so the app is more close to an Android App.

Application Flow

Server Side

First, the server should be initialized. Running the main method in the *ServerApplication* class effects in the instantiation of the ServerApplication object, which represents all models on the server-side. Let this object be called *server.* The *server* is initialized by creating a default board named "Default", and a thread that runs the Runnable *ServerConnectionHandler* is started. This thread is responsible for listening to new connections from clients.

When a client connects, the *ServerConnectionHandler* instantiates representations of the connection and also it's handlers. The connection is represented as a *Connection* object. Each *Connection* has informations about the connected client, such as username, active board, last version of the active board sent to the client, etc. Also, each *Connection* has a thread that is

responsible for checking the version of the Whiteboard in the server-side, and schedule updates of the board to be sent to the client when there are updates available. Also, each *Connection* has a *ConnectionController*, which receives and sends information from clients, and also apply methods (observers and mutators) over the Models. Each client has its own *Connection* and *ConnectionController*, and when the client disconnects, these two are removed from the server, and all threads in them are stopped.

The *ConnectionController* has two threads, one that receives information from the client and warns the server to perform the correct operation, and another one responsible for sending information to the client.

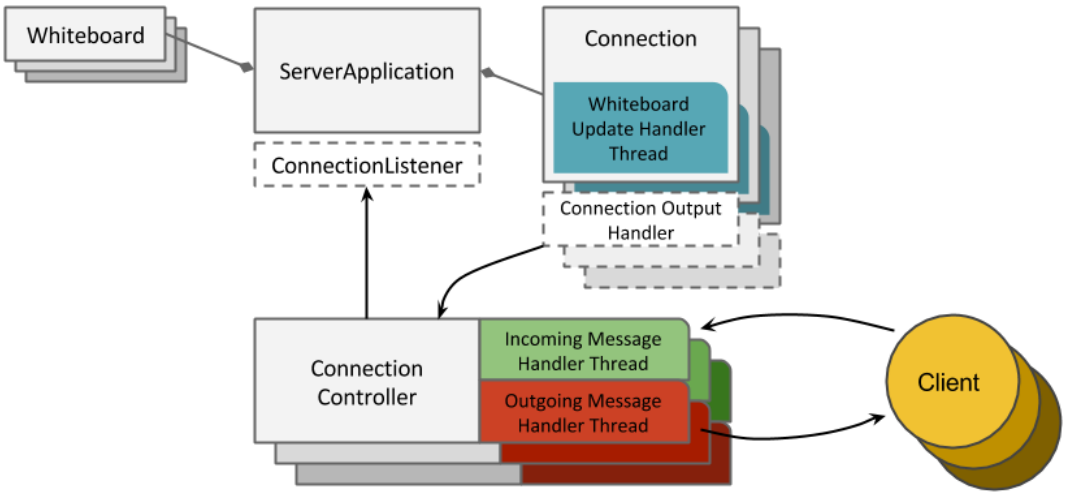

Figure 1: Server Side Application flow

Client Side

To start a *ClientApplication* you need to run the main method in the *Main* class, inside the *client* package. The main method instantiates the *ClientApplication* object, creates the GUI, and it configures the *ClientApplication* by setting the desired username, inputted by the user, and by setting a ClientListener, which in the Java Swing GUI is the *WhiteboardGUI* itself. The *ClientApplication* uses the listener to inform which kind of message it received, and to delegate to the GUI the task of displaying to the user the new information received.

A *ClientApplication* is the model of the client. It has information about the Whiteboard it is working in, and also about itself. Just as the Server, it has two handler threads that acts as a controller, receiving messages from the server, and applying methods over the *ClientApplication* models.

The user can input information through the Controllers of the GUI. For the Canvas, we have different controllers that implement the *ToolController* interface. Each *ToolController* has information about how they should receive the inputs from the user, and what messages should be sent to the server. Also, the controller knows how to paint messages that comes from the server, and the Canvas knows which controller to use, depending on the message the client

receives. The canvas paints in its screen what is represented by the Whiteboard object. When the client decides to paint, change whiteboard, or anything else that is shared between the other client, first it sends the request of what he wants to do, and the action only happens when it receives the "permission" from the server. The server sends the information in an ordered way, so the Whiteboard is always synchronized with the server's.

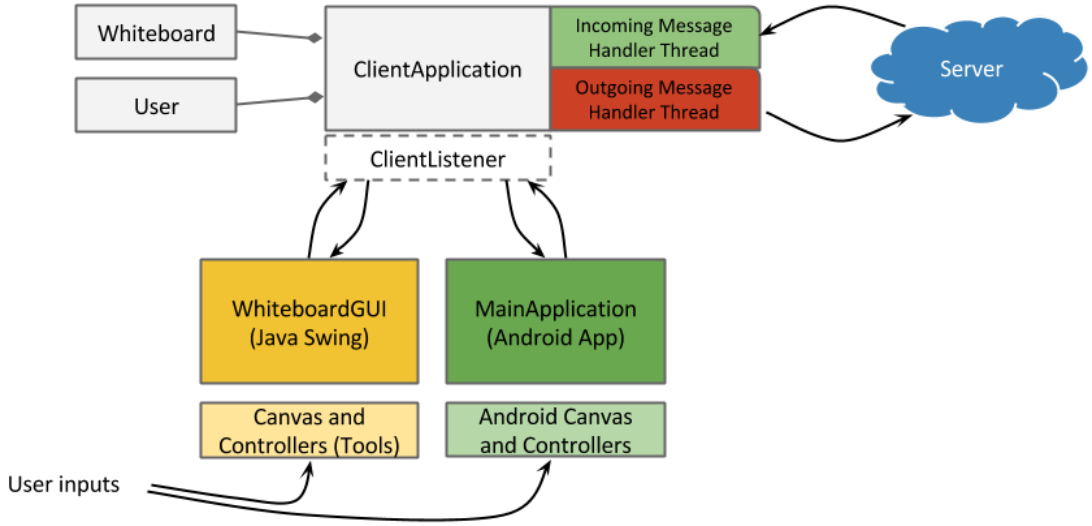

Figure 2: Client Side Application flow

Datatype Design

1. Shared Classes

Whiteboard

A model of a whiteboard. It represents both clientside and serverside models. Each instance of Whiteboard has its own name and version. The drawing in a whiteboard is simply a collection of actions, defined by our protocol. A GUI that wants to draw a whiteboard needs to draw all actions inside a Whiteboard. It is mutable.

Methods

- **Whiteboard(String) : Whiteboard**
  Constructor method. Receives the Whiteboard name as a parameter.
- **getName() : String**
  Return the Whiteboard name.
- **getVersion() : int**
  Return the version of the board.
- **getAction(int) : String**
  Return the action in the position ID passed as a parameter. ID must be non-negative and less than the Whiteboard version.
- **update(String) : void**

Update the model with the action passed as a parameter. Updates the version too. The action passed have to be a valid action defined by the protocol.

User

Represents a User that is using the collaborative whiteboard application. It is mutable

Methods

- **User(String) : User**
  Constructor. Defines a random unique id to this object.
- **User(String, String) : User**
  Constructor. Creates an instance of user. Sets the unique id.
- **getUid() : UUID**
  return the unique id of this object.
- **getName() : String**
  return the name of this user.
- **setName(String) : void**
  Sets the name of this user. param username. the desired username, cannot have spaces.

CWPMessage

Represents messages of the custom protocol defined for this application. CWP stands for Collaborative Whiteboard Protocol. It is an immutable class.

Methods

- **Encode(User, String, String[]) : String**
  Static method. Defines a message as a String. The sender, action, and parameters are passed as arguments to this method.
- **EncodePaintAction(User, String) : String**
  Static method. Defines a message as a String. The sender and the paint action, which is used to paint in a Whiteboard, are passed as arguments.
- **validate(String) : void**
  Validates a message. It checks if the message has a valid syntax. defined by the Protocol.
- **getSenderUID() : String**
  Returns the UID from the Sender.
- **getAction() : String**
  Returns the action represented by this message.
- **getArgument(int) : String**
  Returns a specific argument.
- **getArguments() : String[]**
  Returns an array of all arguments in the current message.
- **getArgumentsSize() : int**
  Returns the number of arguments present in the current message.

- **getPaintAction() : String**

  If the method is a paint action, return the portion of the message that represents the paint action. This means removing the header of the message (such as sender UID).

2. Server Side

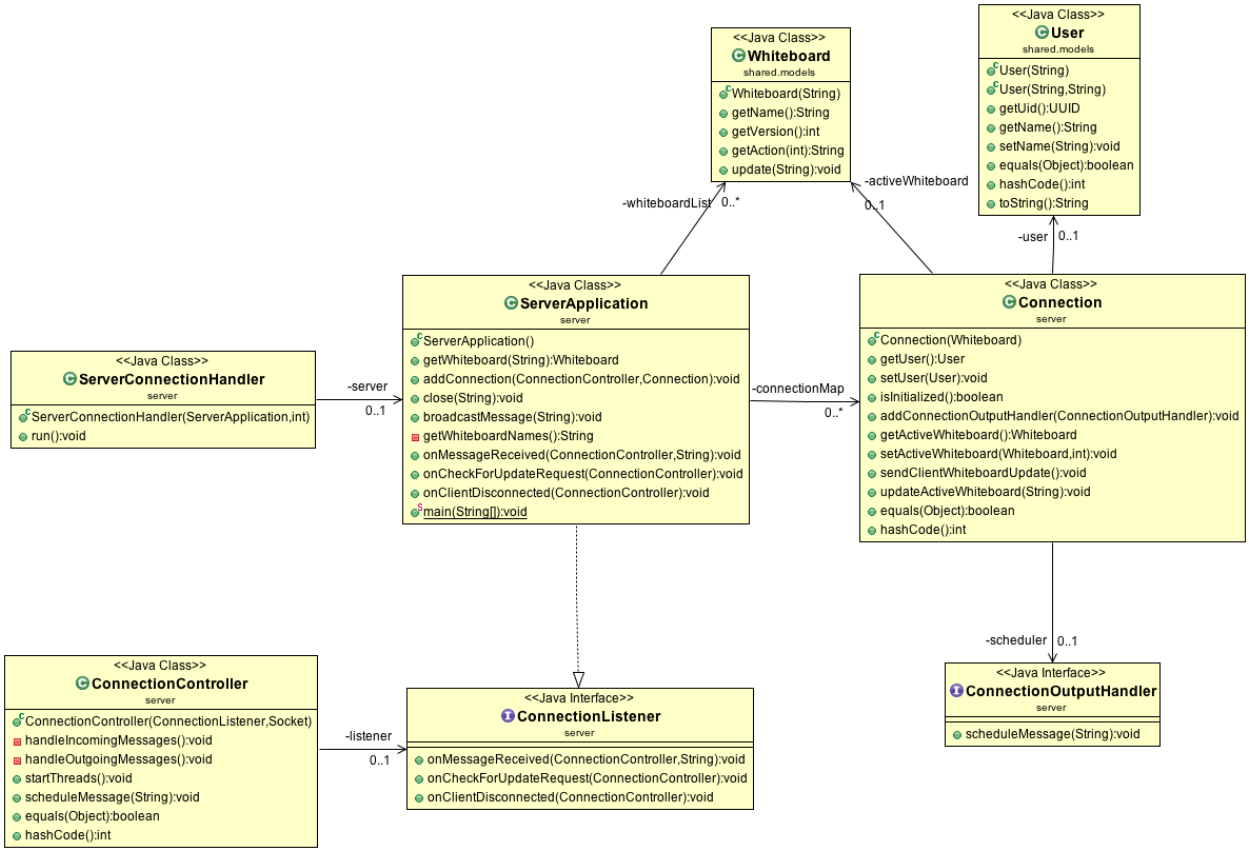

Figure 3: Server Side class diagram

ServerApplication

This class represents the Model of the Server. It has the representation of all connected clients and also all active whiteboards. The server's whiteboards are the official ones, and all clients are always updating themselves to keep consistency with the server's representation. It is also the entry point of the Server application. It is mutable.

Methods

- **ServerApplication(String serverName) : ServerApplication**

  Constructor. Creates other models that are part of the server's representation.
- **main(String[]) : void**

  Runnable method. Runs the server.

- **serve() : void**
  Listens to connections from clients. After receiving a connection, creates a ClientConnection and a ClientHandler to take care of the connection, and starts listening again for new connections.
- **getWhiteboard(String) : Whiteboard**
  Return the Whiteboard with the name passed as a parameter. If a Whiteboard with the passed name doesn't exist, creates a new one, and also sends a broadcast to everyone connected that a new board was created. The name cannot contain spaces.
- **addConnection(ConnectionController, Connection) : void**
  Adds a new Connection to this server.
- **broadcastMessage(String) : void**
  Broadcasts a message to all connected clients.
- **getWhiteboardNames() : String**
  Return a list of names of all the active Whiteboards in the server.

ServerConnectionHandler

Handles incoming client connections. It is responsible for creating a Connection and ConnectionController to handle this new connection. Immutable class. It is a runnable, so it only has the run method.

Connection

Represents a client connection on server-side. This connection has information about the connected client, such as his username, which board the client is working on, and also the last version sent to the client of that board. It is mutable.

Methods

- **Connection(Whiteboard) : Connection**
  Constructor. Receives which server it is working for, and also the active board that should be sent to the client.
- **getUser() : User**
- **setUser(User) : void**
- **isInitialized() : boolean**
  Verify if the connection is initialized. The connection being initialized means that other clients can acknowledge its existence.
- **addConnectionOutputHandler(ConnectionOutputHandler) : void**
  Sets a handler that knows how to send messages to the server.
- **getActiveWhiteboard() : Whiteboard**
- **setActiveWhiteboard(Whiteboard, int) : void**
  Change the client's active board. Also, sets which version the client has.
- **sendClientWhiteboardUpdate() : void**

Sends through this connection an update of the active board. It sends an update iff the last version sent to the client is older than the version in the server.

- **updateActiveWhiteboard(String) : void**
  Update the current active board with an action. This method updates the version of the server's representation of the board.

ConnectionController

Handles the connection of the client. Each ConnectionController handles exactly one Connection. It has three threads: One responsible for receiving and client's messages, other responsible for sending information to the clients, and another one responsible for updates to be sent to the client regarding the active board it is working on. This controller handles all incoming and outgoing packages from client and server. It is mutable, but the only thing that mutates is the queue of messages scheduled to be sent to the client

Methods

- **scheduleMessage(String) : void**
  Schedules a message to be sent to the client.
- **startThreads() : void**
  Start the threads that handles the input and output stream of the ClientConnection.

3. Client Side

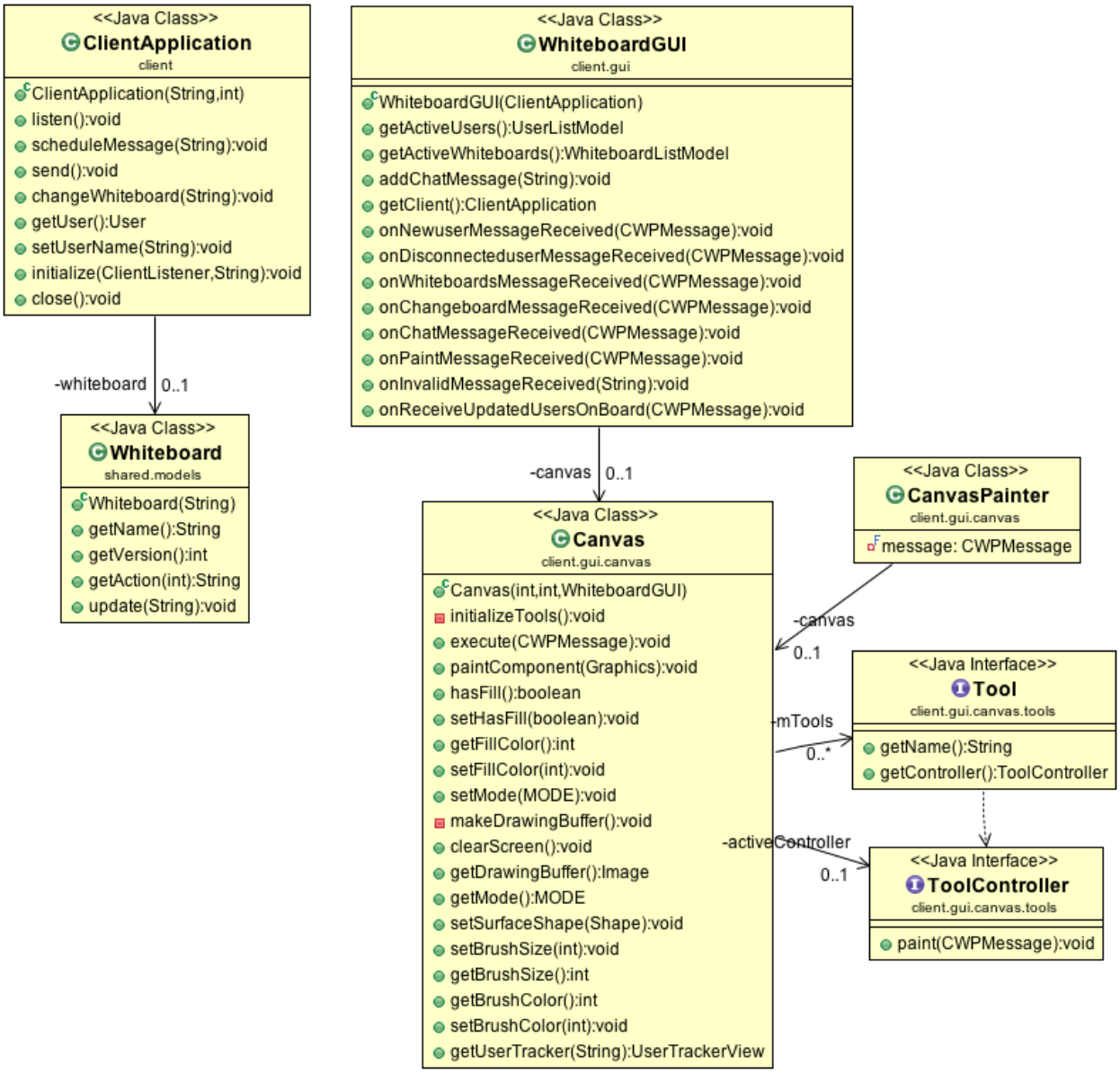

Figure 4: Client Side class diagram

ClientApplication

Represents a client of our collaborative whiteboard application. It is the main entry point of the application, managing the connection with the server. It is responsible for sending and receiving messages written in our protocol to the server. Also, the client should be able to select which board he wants to work on, and also know who is connected to the server. Each ClientApplication has its own username too. The client connects to the ClientApplication through a socket.

Methods

- **ClientApplication(String, int)**
  Constructor. Receives the server ip and the port as parameters.
- **main(String[])**
  Runnable method. Starts the client by creating an ApplicationClient and the GUI.
- **initialize(ClientListener, String)**
  Initializes the client by sending an initialization request to the server.
- **listen()**
  Listen to messages sent from the server. Also, routes the message to the right object depending on its content (for instance, the Whiteboard).
- **scheduleMessage(String)**
  Sends a message to the server. The message content is passed as an argument.
- **changeWhiteboard(String)**
  Sets this class's whiteboard to a new instance of Whiteboard.
- **send()**
  Checks if there is a message to be sent. If there is a message, then it sends it. If there isn't, it blocks the stream.
- **setUserName(String)**
  Sets this class's user's name.
- **close()**
  Close the socket connection.

WhiteboardGUI

Is the user interface of the application on client-side. It has several buttons to control what will be drawn over the canvas.

Methods

- **WhiteboardGUI(ClientApplication)**
  Initializes and sets up the display.
- **getActiveUsers()**
- **getWhiteboardListModel()**
- **addChatMessage(String)**
  Appends a line of user-input text in the chat area pane
- **getClient()**
- **onNewuserMessageReceived(CWPMessage)**
- **onDisconnecteduserMessageReceived(CWPMessage)**
- **onWhiteboardsMessageReceived(CWPMessage)**
- **onChangeboardMessageReceived(CWPMessage)**
- **onChatMessageReceived(CWPMessage)**
- **onPaintMessageReceived(CWPMessage)**
- **onInvalidMessageReceived(CWPMessage)**

- **onReceiveUpdatedUsersOnBoard(CWPMessage)**

    All these methods overriden from the ClientListener interface are callbacks that are called by the client when each operation occurs, changing the GUI accordingly.

Canvas

Represents a drawing surface that allows the user to draw over it with the mouse.

Methods

- **initializeTools() : void**

    Initialize all tools that can be selected to draw over this canvas.
- **execute(String)**

    Execute an action from the Whiteboard representation over the canvas.
- **paintComponent(Graphics)**

    Modified so it can paint the Whiteboard and also have local drawings to show to the user.
- **hasFill()**
- **setHasFill(boolean)**
- **getFillColor()**
- **setFillColor(int)**
- **setMode(MODE)**
- **makeDrawingBuffer()**
- **clearScreen()**
- **getDrawingBuffer()**
- **changeMode(MODE)**
- **getSurfaceShape()**
- **setSurfaceShape(Shape)**
- **setBrushSize(int)**
- **getBrushSize()**
- **getBrushColor()**
- **setBrushColor(int)**

CanvasPainter

Implements Runnable. It is responsible for painting an action of the Whiteboard over the Canvas. It exists so it keeps the GUI single threaded, by putting an instance of this object in the queue of actions to be done by the GUI.

Tool

An interface of a tool that carries controllers to be used by the GUI. May have information of where it should be grouped or displayed in the GUI.

Methods

- **getName()**
- **getController()**

ToolController

An interface of all the controllers that can be attached to the GUI to perform actions over the Canvas. This interface extends MouseListener and MouseMotionListener to recognize mouse actions in the GUI.

Methods

- **paint(String[])**
- **mouseDragged(MouseEvent)**
- **mouseMoved(MouseEvent)**
- **mouseClicked(MouseEvent)**
- **mousePressed(MouseEvent)**
- **mouseReleased(MouseEvent)**
- **mouseEntered(MouseEvent)**
- **mouseExited(MouseEvent)**

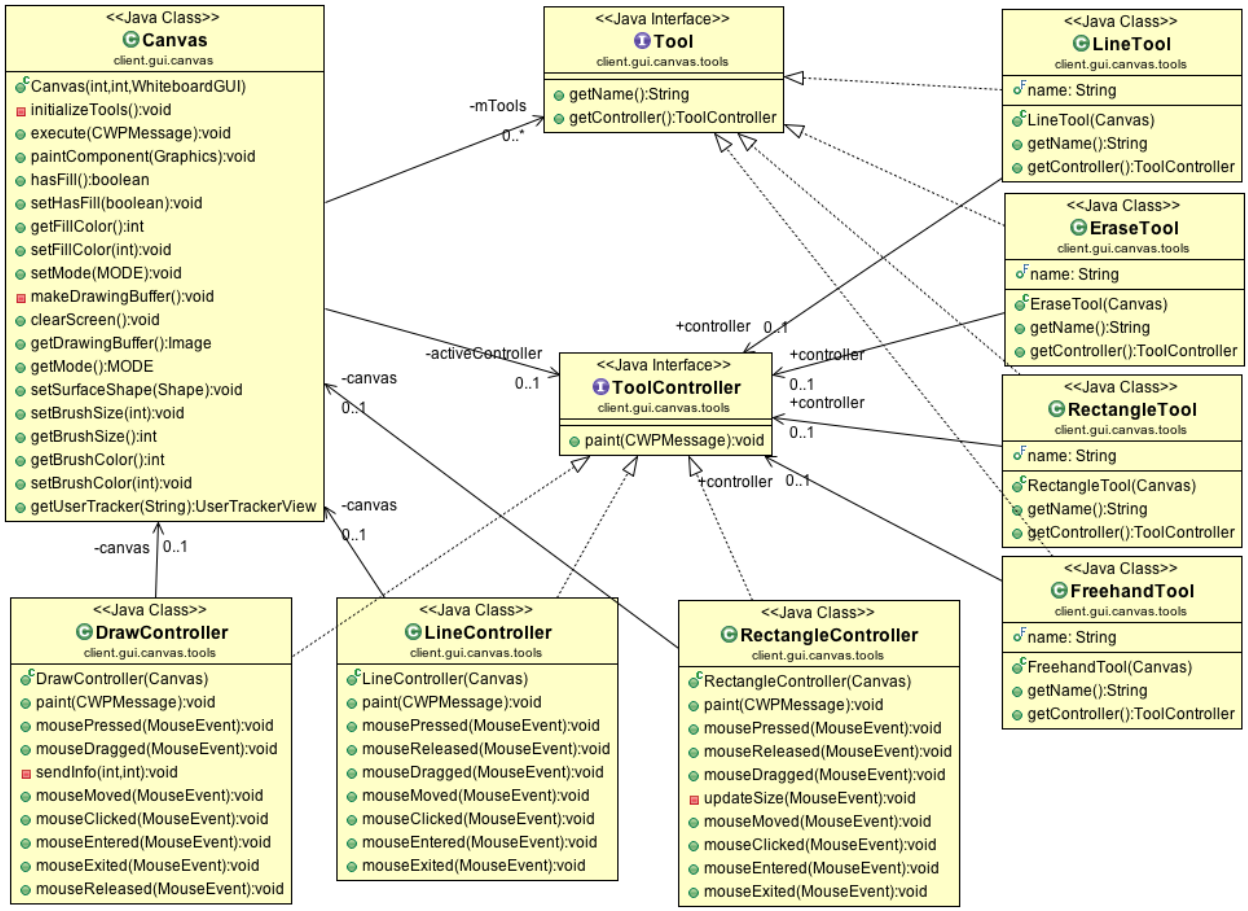

Figure 5: Canvas and Controllers class diagram

Protocol

Each message in our Collaborative Whiteboard Protocol (CWP) is in the following format:

```
UUID action arg0 arg1 … arg x
```

Also, each token is divided by a *SEPARATOR*, defined as a static string *CWPMessage.SEPARATOR*. The *CWPMessage* class was defined before in our shared models in the beginning of this document.

The messages that are valid in our protocol are:

- **uuid initialize uuid username**
- **uuid disconnecteduser uuid username**
- **uuid newuser uuid username**
- **uuid drawline x1 y1 x2 y2 brushSize color**
- **uuid erase x1 y1 x2 y2 brushSize whitecolor**
- **uuid drawrect x1 y1 x2 y2 brushSize brushColor hasFill fillColor**
- **uuid changeboard boardname**
- **uuid whiteboards (boardname)*** -> can have from zero to *n* arguments
- **uuid chat message**

Concurrency Strategy

Server Side

Our models (*ServerApplication*, Whiteboard, and User) are designed for thread safety, by choosing thread-safe datatypes (such as synchronized lists, queues and maps). When a method executes actions that might break the invariant, we perform locking to ensure the representation invariant. For Whiteboard and User in special, we use the Monitor pattern to guarantee thread safety. Also only one thread has access to the output stream, implementing producer-consumer pattern using a Synchronized Queue, where the consumer gets information produced by producers, and send them to clients. The server keeps track of each Connection with a Map from *ConnectionControllers* to *Connections*. This map is also synchronized, since each client has threads that could act over it (for instance, sending a broadcast to all clients connected). Also, the server keeps track of all Whiteboards by using a Synchronized List. To keep consistency of each Whiteboard, the Server adopts a first come first served policy when painting over a whiteboard, and clients receives updates of this same actions in the correct order, which guarantees that every client has the same view of the board. When a client changes its active board, the server sends all the actions performed over the new board to the client. Although we guarantee thread-safety for all classes inside the Server, an instance of *ServerApplication* itself is single-threaded, and is not supposed to share information with other *ServerApplication* objects. This means that each object is a separate server, and don't work as a distributed system.

Client Side

We also guarantee thread safety on client-side by using thread-safe datatypes, such as Synchronized Queue for sending messages just as implemented in the server side, and implementing the monitor pattern when necessary. For the GUI, everything update that needs to be reflected in the GUI uses the Swing Queue to perform operations (with InvokeLater() method), this way all modifications in the GUI are made in a single-thread. Modifications in the model are made by only one thread, which listens to messages from the server.

Testing Strategy

First we perform Unit Testing over the classes that don't have socket dependencies. Theses classes are:

- User
- Whiteboard
- Connection
- CWPMessage

For the *CWPMessage* class in special, we test all the valid messages of our protocol when testing the validate method.

We also perform unit testing over the *ClientApplication* and *ServerApplication* classes, but only for methods that are not related to socket communication.

To test the Client and Server, we then perform Integration Tests. These tests instantiates a Server, and verifies its behavior by sending messages to it, and checking the expected response from the Server. The same thing is made in the *ClientApplication* integration tests. For testing the *ClientApplication* in special, we tested its behavior by verifying if the ClientListener configured to the client responds accordingly for each message sent to the *ClientApplication* object.

Tests related to the GUI were made during development. For instance, when implementing controllers (listeners), the first thing to be done is to check if the listener fires the correct method by creating a mock listener only for testing. We followed the idea of doing little programming, and testing afterwards every time.