Project 2 | Design Milestone

6.005: Software Construction (Fall 2013)

Team Members

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

Datatype Design

1. Shared Models

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

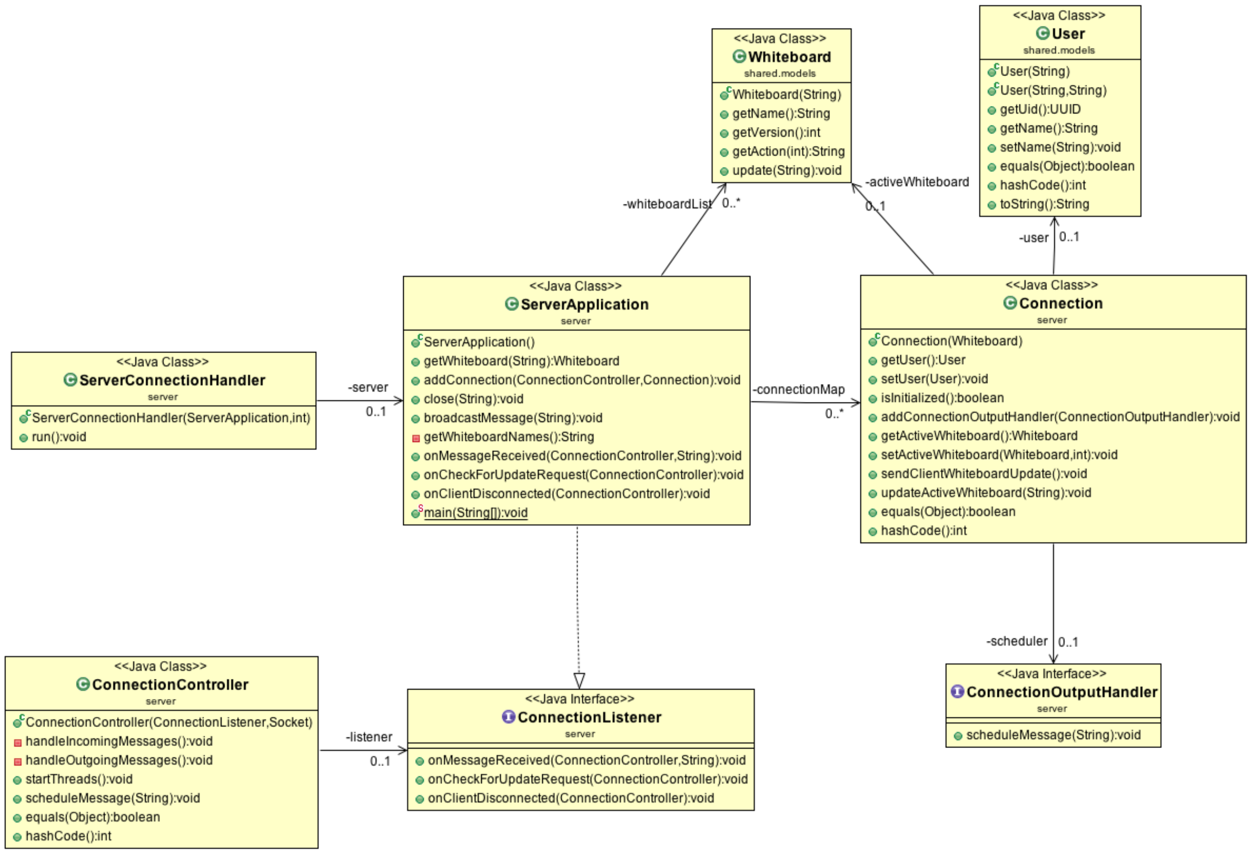
Represents a User that is using the collaborative whiteboard application. It is mutable

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.

2. Server Side



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() : 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.

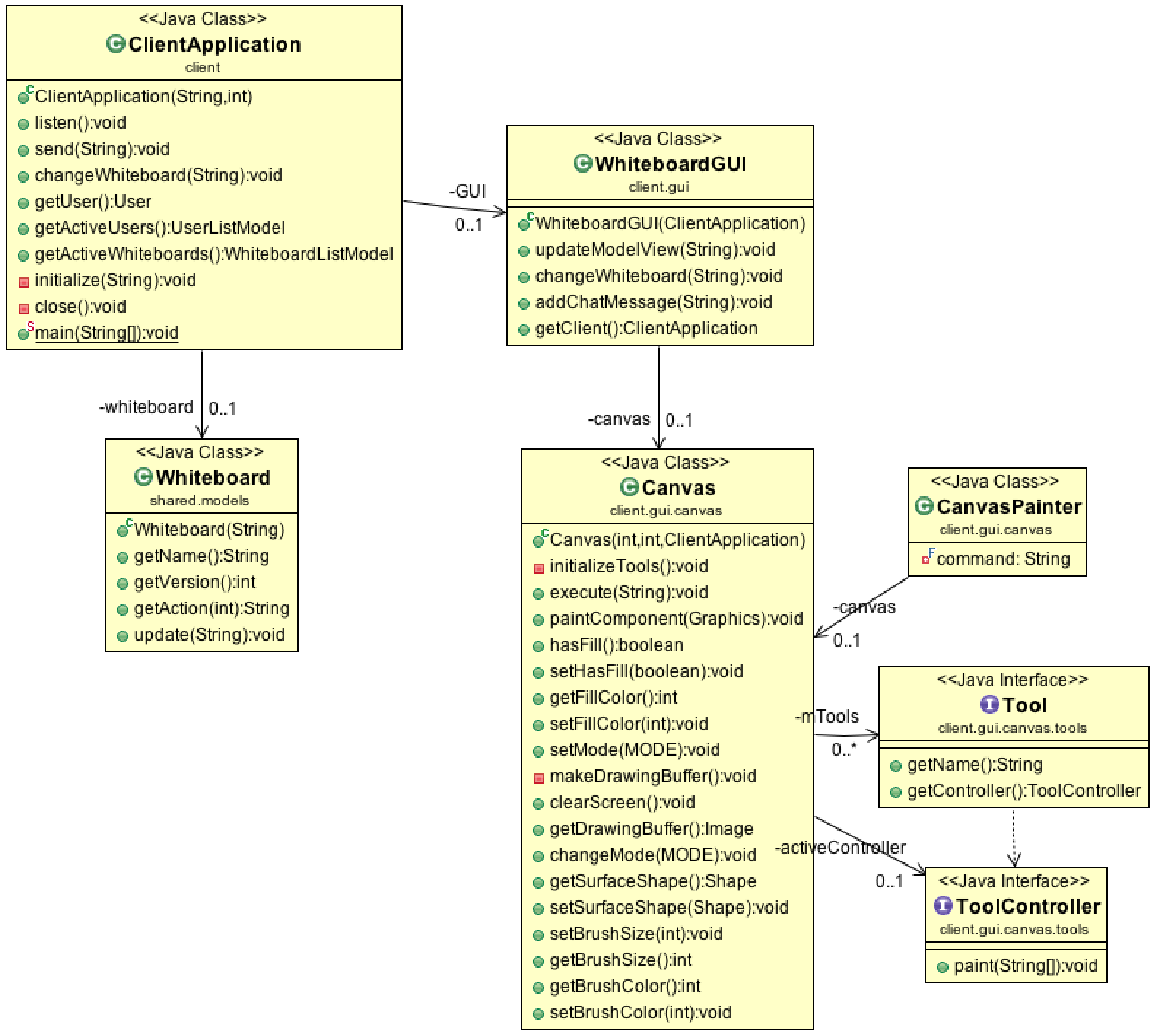
ConnectionController

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



ApplicationClient

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 ApplicationClient has its own username too. The client connects to the ApplicationServer through a socket.

Methods

- **ApplicationClient(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.

- **listen()**
  Listen to messages sent from the server. Also, routes the message to the right object depending on its content (for instance, the Whiteboard).
- **send(String)**
  Sends a message to the server. The message content is passed as an argument.
- **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

- **updateModelView(String)**

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)**

Protocol

Messages sent from the client include the ID of the client prepended to the message (e.g. "uid erase ….") so the server can identify who sent the message.

Our protocol is designed to support the following messages:

Commands sent and received both sides

- erase x1 y1 x2 y2 brushSize
- drawline x1 y1 x2 y2 color brushSize
- drawrect x1 y1 x2 y2 brushColor brushSize fillColor hasFill
- changeboard boardName
- newuser username

Commands sent from the client (to the server)

- initialize username

Commands sent from the server (to the client)

- whiteboards name1 name2 name3 ...
- disconnecteduser username

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, where the consumer gets information produced by producers, and send them to clients.

Client Side

We also guarantee thread safety on client-side by using thread-safe datatypes, 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 (InvokeLater). 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

Our testing strategy will follow the order:

1. User
2. Whiteboard
3. Connection
4. ConnectionController
5. ServerApplication

Here we will start testing our communication with the server. Since all messages are strings, it is easy to test expected input/output from the server.

6. ClientApplication
7. GUI elements

We will do the same thing we did with the server-side on the client-side, testing message passing. Testing the GUI is more difficult, so we will test behavior of some classes (such as canvas), but some things will only be visually tested. For concurrency tests, we will perform manual tests by connecting several clients.