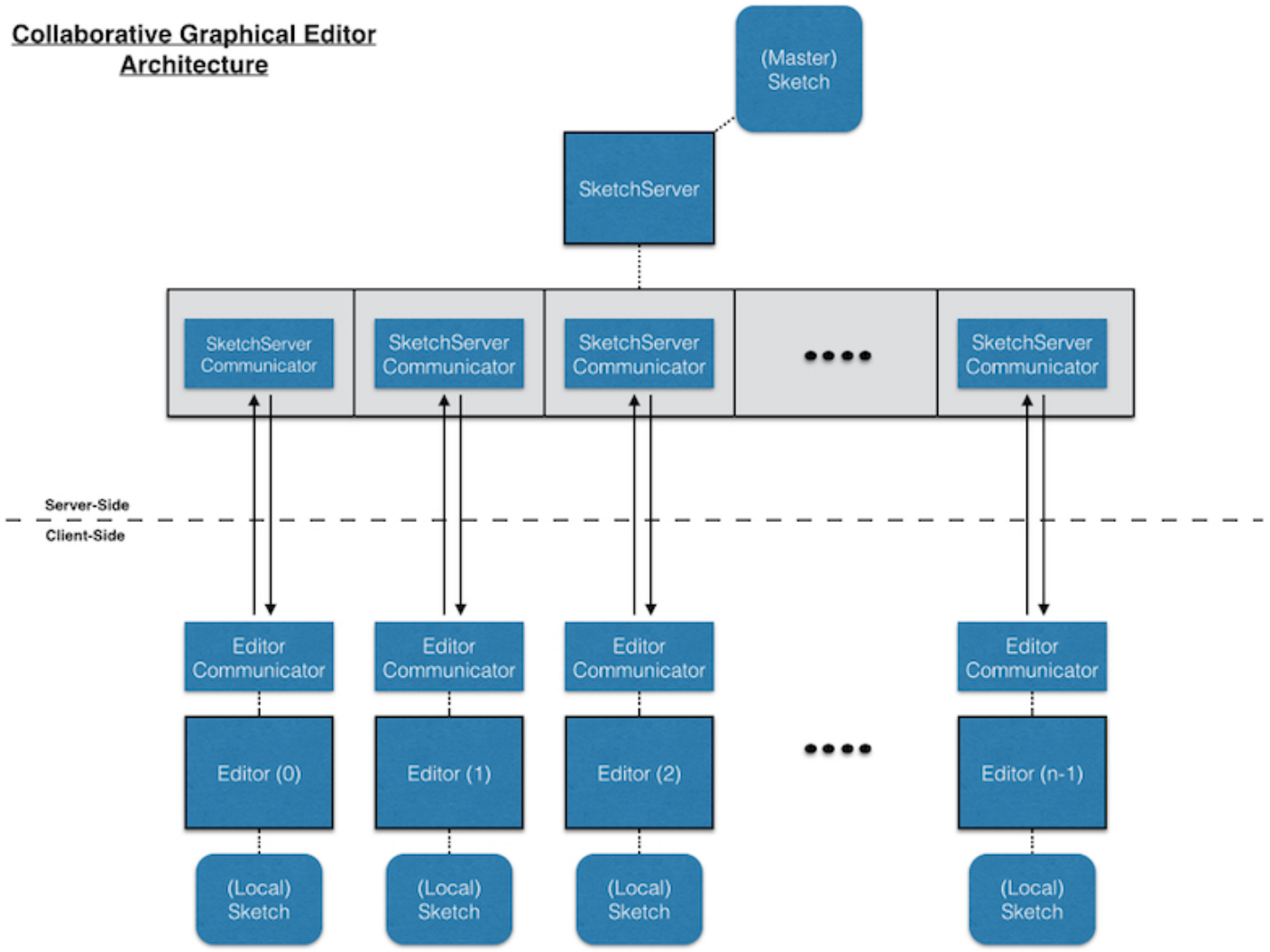
CS 10 PS-6
Collaborative graphical editor

Introduction

As promised, we're going to build a collaborative graphical editor — akin to Google Docs' ability to have multiple simultaneous editors of the same document. In both cases, multiple clients connect to a server, and whatever editing any of them does, the others see. So I can add an ellipse, a friend on another computer can recolor and move it, etc. In addition to being collaborative, the graphical editor is more functional than the short assignment version (and for extra credit, you can flesh it out even more), in that it handles multiple objects at a time, and can draw rectangles and line segments in addition to ellipses.

The basic client/server set-up is much like that of the chat server. Each client editor has a thread for talking to the sketch server, along with a main thread for user interaction (previously, getting console input; now, handling the drawing). The server has a main thread to get the incoming requests to join the shared sketch, along with separate threads for communicating with the clients. The client tells the server about its user's drawing actions. The server then tells all the clients about all the drawing actions of each of them.



There is one twist, to make this work nicely. A client doesn't just do the drawing actions on its own. Instead, it requests the server for permission to do an action, and the server then tells all the clients (including the requester) to do it. So rather than actually recoloring a shape, the client tells the server "I'd like to color this shape that color", and the server then tells all the clients to do just that. That way, the server has the one unified, consistent global view of the sketch, and keeps all the clients informed.

Given that the architecture is much the same as with the chat server (and other client/server set-ups), the bulk of the assignment comes down to establishing the message passing protocols between client/server and ensuring that they all maintain and modify a consistent shared view of a common sketch.

Implementation Notes

A scaffold is provided. It includes a number of files:

- Editor: client, handling GUI-based drawing interaction (similar structure to the short assignment)
- EditorCommunicator: for messages to/from the server
- SketchServer: central keeper of the master sketch; synchronizing the various Editors
- SketchServerCommunicator: for messages to/from a single editor (one for each such client)
- EchoServer: an alternative server useful for development / debugging
- Shape: interface for a graphical shape (with color), with implementations Ellipse, Rectangle, and Segment

Please take the time to familiarize yourself with these files, and read over the whole assignment carefully regarding how the pieces fit together, before even thinking about coding.

Since this code builds on the short assignment, the sample solution to that will be posted on Canvas. If you choose to look at it, then of course you are not permitted to revise your own for resubmission for a possible improved score.

Add your code to the scaffold, and provide additional classes and methods to finish it off. You can change type signatures for some of the "your code here" methods if you want, and feel free to add more; it's just to give the structure. The code isn't particularly complicated, but there are a lot of moving parts, and you need to be clear on the protocol (request a sketch update to the server and act on a command from the server).

You will need to develop your own classes to maintain sketches (the shapes shared among the editors) and to handle messages between the editors and server.

Some thoughts regarding how to approach the implementation.

Editor

- The structure of the `Editor` looks just like that in the short assignment, but now needs to handle multiple shapes, and make and handle requests via the `EditorCommunicator` instead of directly manipulating a shape.
- Creating a new `Rectangle` or `Segment` works much like `Ellipse`: click one corner and drag to the other.
- For simplicity, if you'd like, the process of drawing a shape can be contained just within the editor — when it's complete (button released), tell the server to add it. All the other commands (delete, recolor, move) should be handled as requests to the server.
- NOTE: to run multiple Editors in IntelliJ you must click Run->Edit configurations.., then select Editor in the left pane and check "Allow parallel run" in the upper right

Sketch

- In order to hold the current shape list (either locally for an editor, or globally for the server), I recommend developing an extra class, which for the rest of this write-up I will call "`Sketch`". Up to you how to implement it.
- All editors need to display and refer to the shapes the same way. Thus they have to be referred to by some global "id", with the sketch maintaining the mapping between ids and shapes. One way to ensure consistency of id numbers:
    - The server maintains a "master" version of the sketch, and broadcasts messages to the clients so that their "local" versions are all equivalent.
    - To add a new shape (once finished), an editor client sends a request to the server to add it, giving the info needed for a new instance. The server will then determine a unique id for that shape, and broadcast to everyone an indication to add the shape (again, with all the info needed for the constructor) with the given id. Thus all editors have the same id for the same shape.
    - Delete, recolor, and move operations use the determined id number.
- I think it's easiest if the partial shape currently being drawn is not part of the sketch yet (as discussed above, that happens upon "add", once the mouse is released), but is handled separately.
- The editor and sketch will need to work together to see which shape contains the mouse press, as well as to draw the shapes. Be careful with front-to-back order — look for the topmost containing shape, but draw the bottommost shapes first so that the later ones cover them. If you use a `TreeMap` for the id-to-shape mapping, with an increasing id number for newer shapes, then a traversal of the tree will yield them in order of newness. That's what `TreeMap.descendingKeySet` and `TreeMap.navigableKeySet` do (high-to-low and low-to-high, respectively).
- Because multiple threads will be accessing a sketch (i.e., the different server communicators for the different client editors), the methods that access its shape list will need to be "synchronized".
- `Ellipse` and `Segment` are finished; `Rectangle` should be straightforward if you understand `Ellipse`.

Messages

- As described, actions are sent to the server and then back to the client and the other clients.
- Communication is always via `String`s, using `PrintWriter.println` on one side to communicate with `BufferedReader.readLine` on the other side. You might want to make use of the `toString` methods of the shapes.
- In order to handle a message, it might be useful to define an extra class that takes a `String`-based command (sent via writer to reader) and parses it to determine what action to perform and update the sketch accordingly. I recommend just using a simple format for the messages (i.e., no JSON or XML!), with the various pieces separated by spaces or commas. Then, for example, you can unpack the pieces into an array by `String.split`, and access the pieces by indexing into the array. You can also convert a `String` to an `int` via `Integer.parseInt`. Ex:

```
String[] tokens = String.split("a b 1 2");
=> tokens == ["a","b","1","2"]
=> tokens[0] == "a", tokens[2] == "1"
=> Integer.parseInt(tokens[2]) == 1
```

- When comparing strings (e.g., to decide which action to perform for a message), be sure that you test equality with the method `equals`, not with ==. For example, `tokens[0].equals("a")` would return `true`, but unfortunately `tokens[0]=="a"` would not.
- `Color.toString` is nice for reading but not for parsing. On the other hand, its RGB value (flashback to PS-1!), is an `int`.

Client-server

- The communicators work much like in the chat server — sending and receiving messages on behalf of the client and server, updating the sketch, and, on the GUI side, having the editor update the display (i.e., don't forget to `repaint` as necessary).
- When a new client joins, they need to be filled in on the state of the world — what shapes are already there.
- To help in development, we have provided an `EchoServer` which you can run in place of a `SketchServer`, testing your `Editor` and `EditorCommunicator` before worrying about the server side of things and how to go about handling multiple editors. When the `EchoServer` is running, it will read in input that your `EditorCommunicator` sends and "echo" it back so that your `EditorCommunicator` can read it in. This should help you while developing/testing your message protocol for updating a sketch, and that you are actually updating your local sketch correctly according to received messages.

Exercises

For this assignment, you may work either alone, or with one other partner. Same discussion as always about partners.

1. Finish off the missing pieces in Editor and `Rectangle`, along with a mechanism for handling the shape list (`Sketch`). These can be tested by directly modifying things, as in the standalone editor, before thinking about messages.
2. Devise a set of string-based messages by which the clients and server will communicate drawing actions, along with a mechanism for parsing these messages and invoking the appropriate actions to update the shapes. The methods are up to your design, but should be fairly straightforward reflections of editor commands to add, delete, move, and recolor.
3. Use the `EchoServer` to test your classes, before dealing with the actual `SketchServer`. By using the `EchoServer`, you essentially have a single-machine drawing program which acts on echoed messages that it sends out and gets back, rather than sending/receiving messages via a `SketchServer`. This should allow you to focus on

the client-side part of this lab and make sure it is working before you try to implement the server-side components.
4. Now all that is left to implement is the `SketchServerCommunicator`. There isn't much code to write but you do have to do a few things to handle multiple instances of editors connecting (e.g., telling a client the current state of the world when they first connect, and getting and handling messages from the client). Now hook them up and let them go. Start up a server instance first. Then start an editor. Test on your own machine (set serverIP to "localhost"), and with a friend (give them your IP address or vice versa). Take a screenshot of a collaboratively edited sketch. If you're working independently, you can just have two Editors running on your own machine (i.e., self-collaboration).
5. Consider possible multi-client issues. What unexpected/undesired behaviors are possible? Can you actually make any of them happen? Where and why do you use `synchronized` methods; will they handle everything? Write a short (approximately a paragraph) response.

You may obtain extra credit for extending and enhancing the editor. Only do this once you are completely finished with the specified version. Make a different file for the extra credit version, and document what you did and how it works. Some ideas:

- Support other types of drawing, e.g., "freehand", constructing a list of segments from point to point as the user drags the mouse.
- Support other common graphical editor commands, e.g., bring to front and send to back (note that you'll need to keep a separate list of the drawing order), group and ungroup, undo and redo, etc.
- Allow a client to "lock down" a shape, so that nobody else can modify it until the lock is released. (Draw such shapes differently to distinguish them.)
- Set up an access control, so that a client can specify with what other clients they are willing to share a sketch.
- Make an Android client.

Submission Instructions

Submit your completed versions of all the classes, thoroughly documented. Also submit a snapshot of a collaborative client-server drawing session (can be collaborative with yourself, but with two different clients running), and a document with your brief dicussion of synchronization issues.

Grading Rubric

Total of 100 points

Correctness (70 points)

5 Rectangle
10 Event handlers
5 Adding to and removing from shape list
5 Identifying selected shape in shape list
5 Updating (moving and recoloring) shapes in shape list
10 Drawing shapes and partial shapes
5 Sending strings between clients and server, using EditorCommunicator and ServerCommunicator
10 Composing messages, and decomposing and acting on them
10 Run loops in ServerCommunicator and EditorCommunicator (5 each)
5 Consistency of sketches across clients and server, including new clients

Structure (10 points)

4 Good decomposition of methods
3 Proper use of instance and local variables
3 Proper use of parameters

Style (10 points)

3 Comments for new methods (purpose, parameters, what is returned)
4 Good names for methods, variables, parameters
3 Layout (blank lines, indentation, no line wraps, etc.)

Testing (10 points)

7 Snapshot of a collaborative drawing
3 Discussion of multi-client