6.005 Elements of Software Construction
Fall 2008
Project 3: Instant Messaging

Problem

Instant messaging (IM) is a staple of the web and has been around almost since its inception, starting with simple text-based programs like talk and IRC and progressing to today's GUI-based IM clients from Yahoo, Microsoft, AOL, etc. In this project you will design and implement an IM system, including both the client and the server. The following characteristics constrain the design space of an IM system:

- **Real-time communication.** An IM conversation happens in real time: one person types some text, presses "enter," and the other person (almost) immediately sees the text.
- **Number of parties.** An IM conversation can happen between two or more people. Some systems only allow two people to communicate; others allow more than two

people. Most systems allow a person to be involved in multiple conversations at the same time.
- **Based on typed text.** The main mode of communication is via text, as opposed to voice or video.
- **Connected over a network.** The parties involved in the communication may be in physically remote locations, and are connected over the internet.

Your task will be to design an instant messaging system with the above properties, as well as additional properties that you will incorporate into your design. This system will include a server component that handles the transfer of messages and other data, and a client component with a graphical user interface.

Purpose

The purpose of this project is twofold. First, you will learn several Java technologies, including networking (to support connectivity over a network), sockets and I/O (to support real-time, text-based communication), and threads (to support two or more people communicating concurrently). State machines may be useful to specify certain aspects of the system's behavior.

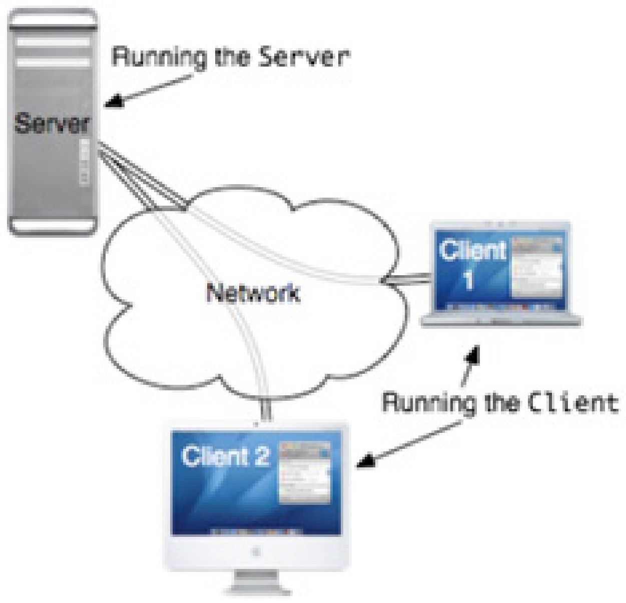
Second, this project will introduce you to the state-of-the-art for enabling human-computer interaction: graphical user interfaces. You will:

- become familiar with Swing, a graphical user interface (GUI) toolkit for Java, that is similar to many other such toolkits;
- learn important GUI programming concepts, including the notion of a *view hierarchy* and the *model-view-controller* design pattern;
- use event-based programming and the publish-subscribe pattern;
- use object modeling notation to explore and express these structures;
- and confront user interface design challenges.

Throughout the project, you will need to design and implement mutable datatypes, paying particular attention to their specifications and how they interact with one another.

Specification

Implement an IM system in Java with the following properties:



- **Client.** The client is a program that opens a network connection with the IM server at a specified IP address and port number. Once the connection is open, the client program presents a graphical user interface for performing the interactions listed below.

- **Server.** The server is a program that accepts connections from clients. A server should be able to maintain an unlimited number of open client connections, and clients should be able to connect and disconnect as they please.

  The server is responsible for managing the state of both clients and *conversations.*

- **Conversations.** A conversation is an interactive text-exchange session between some number of clients, and is the ultimate purpose of the IM system. The exact nature of a conversation is not specified (although the hints section details a couple possibilities), except to say that it allows clients to send text messages to each other. Messaging in a conversation should be instantaneous, in the sense that incoming messages should be displayed immediately, not held until the recipient requests them.

- **Client/server interaction.** A client and server interact by exchanging messages in a protocol of your devising — the protocol is not specified. Using this protocol, the user interface presented by the client should:

  - Provide a facility for seeing which users are currently logged in;
  - Provide a facility for creating, joining and leaving conversations;
  - Allow the user to participate in multiple conversations simultaneously;
  - Visually separate messages of different conversations (e.g., into distinct windows, tabs, panes, etc.);
  - Provide a history of all the messages within a conversation for as long as the client is in that conversation;
- **No authentication.** In a production system, logging in as a client would require some form of password authentication. For simplicity, this IM system will not use authentication, meaning that anyone can log in as a client and claim any username

they choose.

Tasks

1. **Team preparation.** Meet with your team during the project work period on November 19 and complete the lab on team building.

2. **Individual preparation.** Complete the lab on networking by checking out `friendly` from your personal SVN repository. You may also complete the optional Swing lab by checking out `guiwords` from your personal SVN repository (or you may do this lab with your group). These labs are independent of each other.
3. **Abstract design.** Define a precise notion of *conversation* in your IM system. See the hints on how to do this. Construct an object model that captures the essential concepts of instant messaging from a problem perspective, and their relationships to one another. Explain the important modeling decisions you made. Describe alternatives to particular decisions and justify your choice.
4. **Client/server protocol.** Design a set of commands the clients and server will use to communicate, allowing clients to perform the actions stipulated by the specification. Create a specification of the client/server protocol as a grammar or state machine. Describe possible protocol messages, the state of the server, and the state of the client (if it stores any).
5. **Usability design.** Sketch your user interface and its various screens and dialogs. Use these sketches to explore alternatives quickly and to plan the structure and flow of your interface. *Sketching on paper* is recommended. Turn in the sketches you decided to go with, along with commentary as needed to explain non-obvious parts. Briefly point out the merits of your design.
6. **Code design.** Design your program with a module dependency diagram that includes all model, view, and controller classes. Explain important design decisions, and justify your choices with reference to specific alternatives for particular decisions. Your design should minimize the risk of concurrency bugs (like race conditions and deadlocks) and should support easy unit testing of your modules.
7. **Testing strategy.** Devise a strategy for testing your IM system. Describe what automated tests you will use, and what manual tests you will perform. Since UI front-end testing is often most easily done by hand, documentation of your strategy is especially important. As you think about how to test your program, you are likely to find that you want to revisit your code design (for example, to make a cleaner API to permit unit testing independently of the GUI).
8. **Implementation.** As always, your code should be clear, well-organized, and usefully documented. See the hints for further suggestions.
9. **Testing.** Execute your testing strategy, using JUnit and by performing manual tests of

the GUI. In your report, document the results of your manual tests.

10. **Reflection.** Write a brief commentary describing what you learned from this experience:

    - What was easy?
    - What was hard?
    - What was unexpected?
    - What would you do differently in designing the chat system if you were to do it again?

Infrastructure

Use the networking lab to learn about network I/O in Java, and the Swing lab for details on GUI programming.

No initial code is provided for this project. However, two *runner* classes are provided with `main` methods you should fill in:

- Running `main.Client.main(String[])` must start an instance of your GUI chat client.
- Running `main.Server.main(String[])` must start an instance of your chat server.

You should consider using packages other than `main` to organize your code.

Deliverables and Grading

There are *three* deadlines for this project.

For the first deadline, your deliverables are:

- the team contract;
- the abstract designs and discussion;
- the client/server protocol;
- and the usability design.

This design deliverable should be submitted by committing one PDF to the `deliverables` folder of your project repository. For your user interface sketches, you should either scan the files and place them in your PDF, *or* you may hand in paper copies of the sketches (keep and use your originals!) to a TA.

For the second deadline, you will meet with your TA, and your deliverables are:

- the code designs and discussion;
- the testing strategy;
- and a demo of *some* working portion of the project that demonstrates significant effort towards understanding a critical or high-risk area of the design.

The code designs and testing strategy must be submitted by 11am on December 3 as one PDF in the `deliverables` folder of your repository. The demo will take place at the meeting with your TA.

Your demo might show, for example, a basic server that sends and receives messages but without a GUI client (see the the hints about `telnet`). Or you might have a working basic GUI with no server backend but a simple API for connecting to one. Talk to your TA beforehand if you are unsure about what is sufficient.

The meeting will also include discussion of the design deliverables from the first two deadlines.

For the third and final deadline, your deliverables are:

- the implementation;
- the tests;
- the testing report;
- and your reflections on the project.

The report and reflections should be committed as one PDF in the `deliverables` folder.

The grading breakdown is as follows:

- 30% for the abstract design, protocol, and usability design
- 25% for the code design, initial demo, and testing strategy
- 35% for implementation and testing
- 10% for reflection

Awards

The course staff will judge and award prizes to teams whose instant messaging systems embody exemplary design and implementation.

You may **submit your project for prize consideration on Monday December 8 during lecture time, 11am-12:30pm, in 32-G825**. Your team will give a 5-minute presentation to the course staff in which you demonstrate your system and describe its design. You must commit your work (up to that point) to Subversion by 11 am on December 8. You are **not** required to give this presentation (but then you won't win anything, either). And everyone can continue to work on the project until the final deadline, but only the work demonstrated in this presentation will be considered for prizes.

Serious award contenders should consider going above and beyond the required specification to implement their own extensions.

You might add standard instant messaging features like away messages, auto-replies, offline messaging, password-protected accounts, user icons, graphical emoticons... or you might integrate voice chat, a shared whiteboard, encrypted conversations with perfect forward secrecy, or something as yet unheard of!

Hints

Defining a conversation. Part of your job is to determine what a conversation means. For example, does a conversation have a name, and can other users join the conversation by specifying the name? Is it like a chat room, that people can enter and exit? In that case, can a conversation be empty (a chatroom can), waiting for users?

Or is a conversation more like a phone call, where a person "dials" another person? In that case, can the receiving party deny the conversation?

However you define a conversation, remember to *keep it simple for your first iteration.* You can always extend your program with interesting ideas if you have time left.

Designing a protocol. You must also devise a client/server protocol for this project. You should strongly consider using a text-based protocol, which may be easier for testing and debugging.

Services that use plaintext protocols — e.g. HTTP or SMTP — can talk to a human just as well as another machine by using a client program that sends and receives characters. Such a client program already exists on almost all operating systems, called `telnet`. You can run `telnet` by opening a command prompt and typing `telnet` *`hostname port`*. The lab gives you some experience with using telnet.

Handling multiple clients. Since instant messaging is useless without at least two people, your server must be able to handle multiple clients connected at the same time. The Friendly server you'll develop in the lab gives you some starting code, but note that Friendly doesn't need its clients to interact or share any state. Your server will certainly need to do that. One reasonable design approach follows the Friendly model (using one thread for reading input from each client) but adds a central state machine representing the state of the server (using one more thread, to which each of the client threads pass messages through a shared queue).

Read the socket documentation referenced in the lab to understand how network sockets operate in Java. Consider how, for example, the server will write to clients while at the same time awaiting messages from them.

Design for safe concurrency. In general, making an argument that an implementation is free of concurrency bugs (like race conditions and deadlocks) is very difficult and error-prone. The best strategy therefore is to design your program to allow a very simple argument, by limiting your use of concurrency and especially avoiding shared state wherever possible. For example, one approach is to use concurrency only for reading sockets, and to make the rest of the design single-threaded.

And note that, even though user interfaces are concurrent by nature, **Swing is not thread safe**. Understand what code will run in the main thread, threads you explicitly spin, or the Swing event dispatching thread. Recommended reading: Threads and Swing.

Design for testability. To make it possible to write unit tests without having to open socket connections and parse streams of responses, you should design your state machine(s) in such a way that they can be driven directly by a unit test -- either by calling methods, or by putting messages into a queue read by the state machine's thread.

Testing GUIs is particularly challenging. Follow good design practice and separate as much functionality as possible into modules you can test using automated mechanisms. You should maximize the amount of your system you can test with complete independence from any GUI.

Another useful testing technique is the idea of a *stub* (method stubs, mock objects). To test one component of your system in isolation, you can create trivial implementations of the other components with which it is coupled. This might allow you to test your server without opening network connections, or to test your client backend with automated rather than GUI tests.

Implementation. Develop in iterations. Focus on important modules first, and defer making cosmetic improvements to your user interface until after all the code is well-organized and thoroughly tested. Make use of assertions.