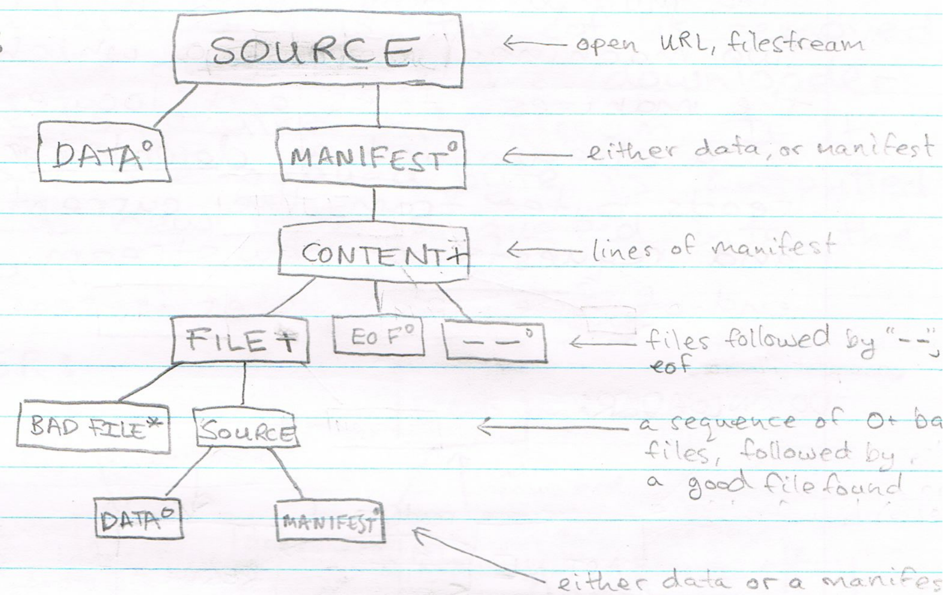
June 19,
Scott Yo

6.005 - Elements of Software Construction

DESIGN PROJECT ONE: Multipart File Transfer

1) Design a grammar for the manifest of a multipart file transfer.

JSP:



Textual:

Source ::= Data | Manifest
Manifest ::= (File+(-- | EOF))+
File ::= BadFile* Source

Data ::= byte*
BadFile ::= URLException

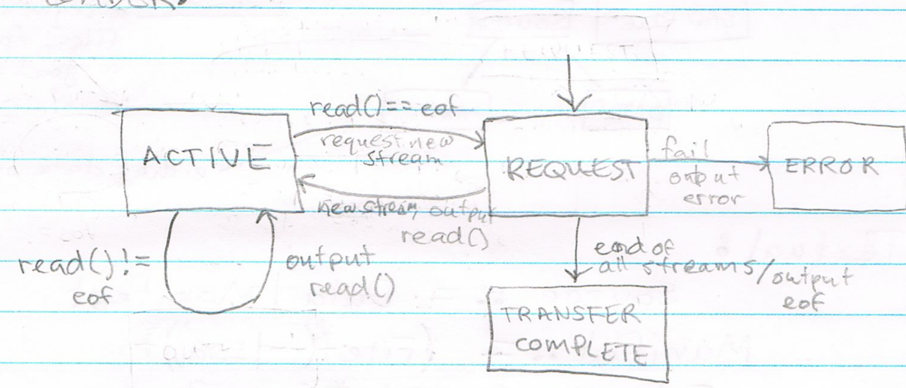
In plain English, the stream is defined as a source file, which is either data or a manifest. If it is a manifest, then it contains repetitions of one or more files separated by either -- or end-of-file terminals. A file consists of zero or more invalid files followed by a source.

Hilroy

2) Design a grammar/state machine for the downloader.

The multipart downloader will have two machines a <u>parser</u>, which manages the manifest files and locates the next data stream, and a <u>downloader</u> which reads bytes from the current stream and requests a new stream if an end-of-file character is reached.

DOWNLOADER:

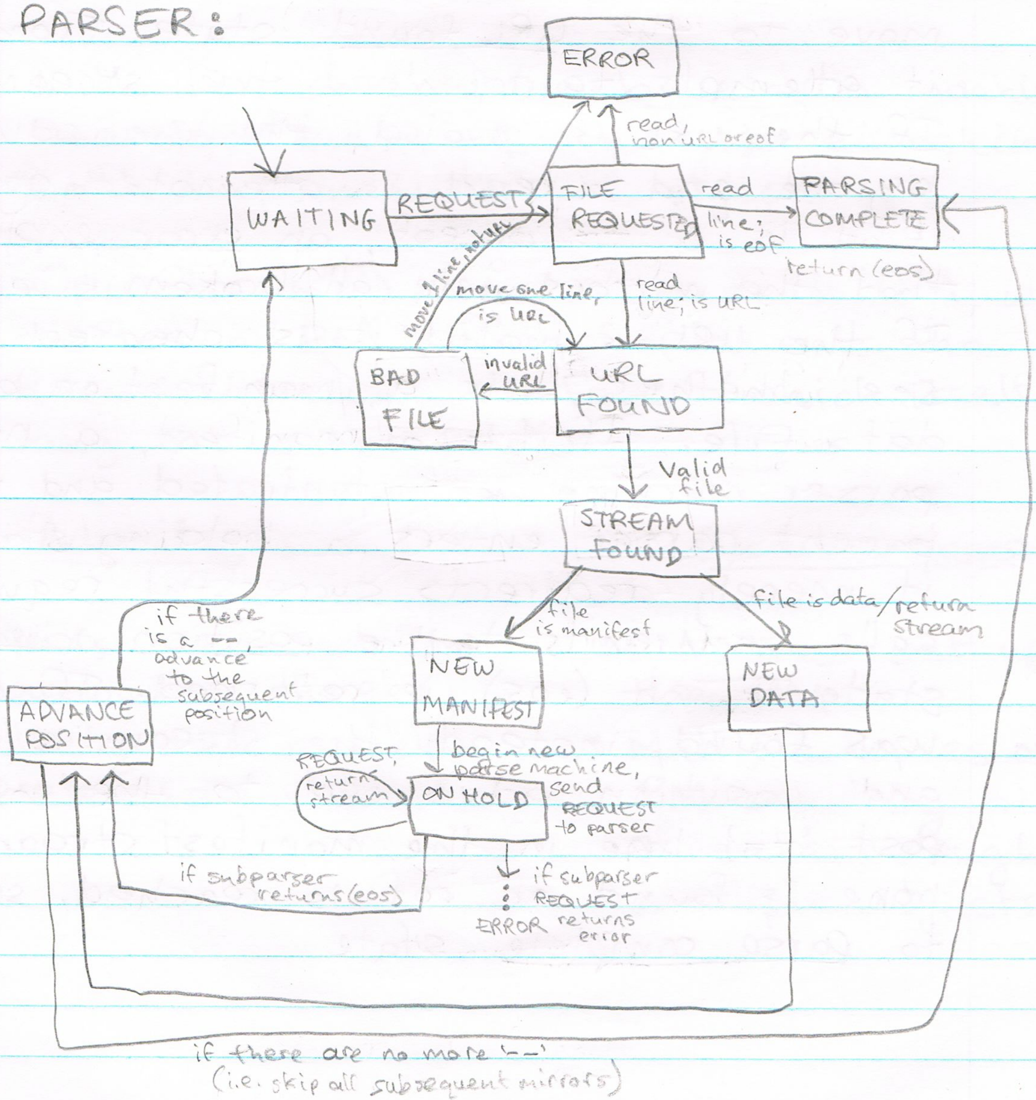


The grammar for this machine is:

(Read*(Error|Complete))
Read ::= byte*eof

The state machine has four states. It begins in <u>request</u> where it queries the parser for a new stream. If an error occurs, the downloader fails and outputs an error. If no files are left, it adds eof to the stream queue and completes. If a new stream is found it adds it to the input stream and moves to the <u>active</u> state. Each call to read invokes the downloader which checks if the queued symbol is an eof character. If it is, the eof is removed from the queue and the downloader <u>requests</u> another file stream. If it is not, the queued byte is outputted and a new byte is queued into the stream.

PARSER:

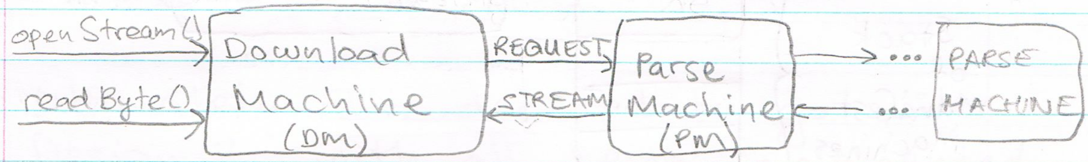


Hilroy

The parser machine is more complex. It is initialized with a manifest stream at the first position, where it attempts to read the next line. From this file requested state it can do one of three things, switch to parse complete iff the next line was eof. From this state all request inputs return end-of-stream. It can return an error if the read was impossible, or it can move to the URL found state. From here it attempts to download the stream. If the URL is invalid, the line advances by one and a read is attempted again, if it is a -- or eof, an error explaining that the mirrors were all broken is reported. If the URL is valid it is checked to see whether it is a manifest or a data file. If it is a manifest, a new parser machine is instantiated and the parent parser enters a holding state where it merely redirects successful request calls and exits to the position advance state if an (eos) is returned. If data was found, instead, the stream is returned and position advanced to the next post '--' line in the manifest stream. If none is found or eof is reached, switch to parse complete state.

3) Code Design: Compare and Contrast two possible implementations

Design One:

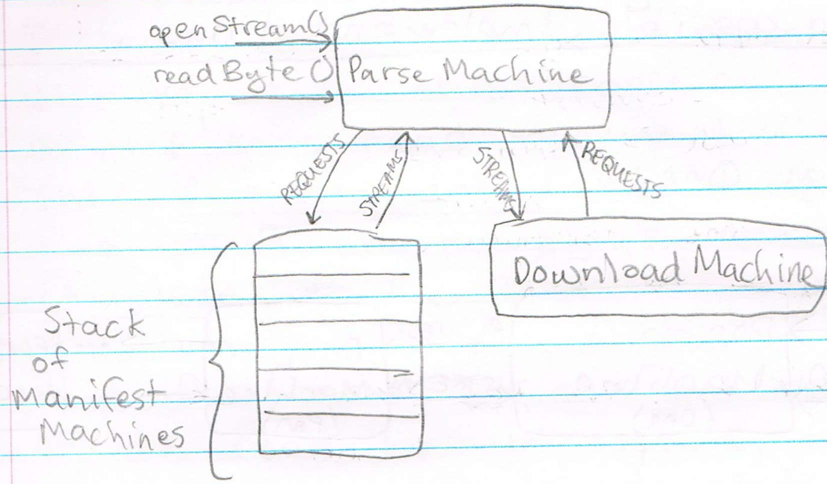


With this design, we have two machines, DownloadMachine and ParseMachine both implemented as objects.

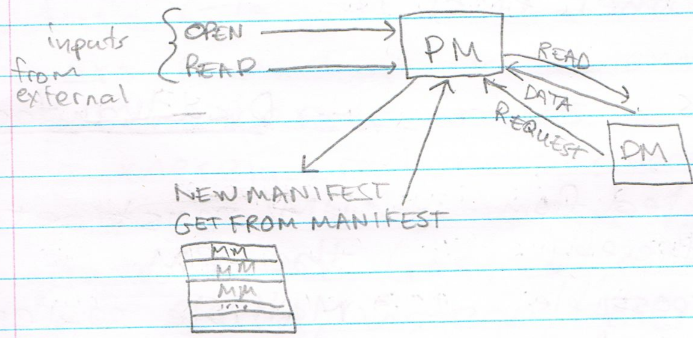
OpenStream() creates a new DM, whic creates a PM which implements the RequestStream interface (which itself may call other PMs).

Advantages	Disadvantages
• PM is separated from downloading, thereby allowing other possible protocols to interface	• DM must set up the PM
•	• Multiple copies of PM requiring pipin of data through multiple objects

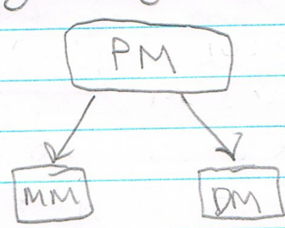
Design Two:



This design splits the ParseMachine into a ParseMachine and a stack of Manifest Machines. The ParseMachine uses a downloader and manifest interface which are implemented by DM and MM. Here are the defined actions:



Dependency Diagram:



Advantages

- Better modularity, can easily exchange different manifest grammars or downloader functions w/o affecting other modules
- No duplication of PM

Disadvantages

- More complicated design.

Ultimately, I've decided to go with Design 2, with all being implemented as Machine-as-Object design patterns.

Textual Specifications:

ParseMachine

→ Contains a stack of machines which implement Manifest interface meaning they have the get() function which returns a URL

→ Also contains a downloader which implements the Downloader interface, which has the read() function and add() function

→ Responsible for (a) determining whether a stream is a manifest and, if so, building a new MM and pushing it to the stack

→ Responsible for taking REQUEST flags from DM and adding a new stream (if available)

→ Responsible for popping a manifest when it returns MANIFEST_FINISHED

Hilroy

ManifestMachine:

Provided with get() MM does one of four things:

- → If the next line is valid, return the URL
- → If it is invalid, ~~was~~ repeat until you get to -- or EOF
- → If --, throw an error
- → If EOF, return MANIFEST_FINISHED flag

DownloadMachine:

Two methods

- → read() returns the first byte in the queue and dequeues it. If it is an EOF, returns REQUEST_STREAM flag.
- → add() queues a new stream

Tests

Given the relatively simple nature of the interface, all unit tests were done on DownloadMachine and ManifestMachine

Reflections

Overall the project wasn't too difficult. In retrospect DM was unneccesary, as PM and MM did virtually all the implementation

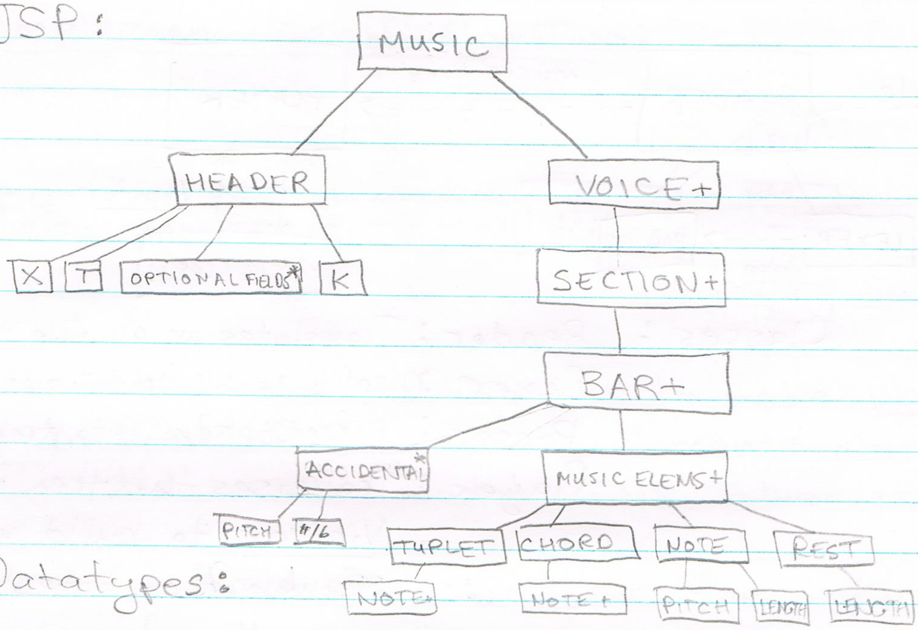
Aug 2, 201
Scott You

6.005 - Elements of Software Construction

DESIGN PROJECT TWO : ABC Music Player

1) Design a grammar for the ABC music notation

JSP:

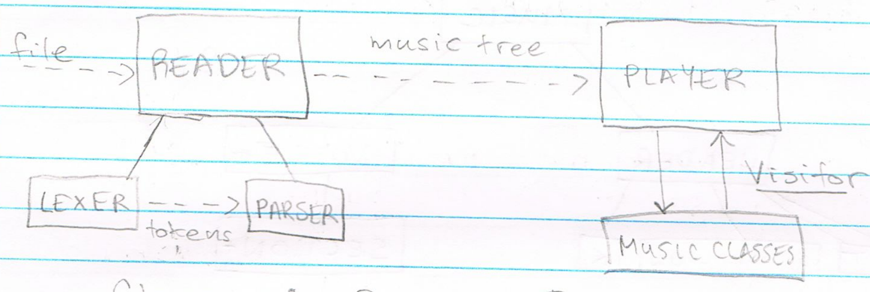


Datatypes:

MUSIC consists of 1 HEADER and 1+ VOICES.
VOICE consists of 1+ SECTIONS
SECTION is either a plain SECTION or a REPEAT
and both consist of 1+ BARS
BAR consists of ACCIDENTALS* and MUSIC ELEMS
ACCIDENTAL consists of 1+ (PITCH and ^, _, =, ^^, _
MUSIC ELEM is either a TUPLET, CHORD, NOTE or REST
TUPLET is either 2, 3 or 4 NOTES
CHORD consists of 1+ NOTES
NOTE consists of a PITCH and a LENGTH
REST consists of a LENGTH

2) Code Design

Implementation Design One:



Classes: Reader : Translates an abcfile to a tree
Lexer : splits text into tokens
Parser : parses tokens into tree
Player : Traverses the tree as a Visitor to build up the sequence of notes

Datatypes: Voice, Section, Bar, Elem, Pitch, Accidental
(classes)

Section → Repeat

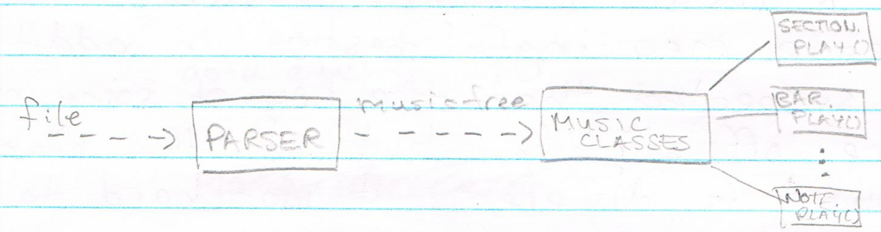
Elem → Tuplet, Chord, Note, Rest

The primitive, Length will just be an int.
Use Factory on Pitch
Accident is an enum containing SHARP, FLAT, DOUBLESHARP, DOUBLEFLAT, NATURAL

Pros: By using the visitor pattern, all code related to playing the notes can be omitted from the classes themselves, therefore preserving there usefulness if we no longer wish to use the MIDI API

Cons: Visitor requires public accessors and therefore exposes the internal workings of the Music datatypes.

Implementation Design Two:



Here the major difference is to make play() a method of each music elements' class, as I did in the Sudoku CNF Solver exploration.

Pros: Without public accessors the interface for the music classes is considerably cleaner and less likely to be abused.

Cons: The design is much harder to port into other music APIs since the sequencing implementation is distributed over the music datatypes themselves.

Conclusion: I will use Design One since it will enhance maintainability and readability at minimal cost.

REFLECTIONS

This design was more challenging because the nature of the MIDI Player makes it harder to compare against a benchmark of a different sequencer, as a result I found it hard to create meaningful tests. In addition, I suspected that the key of some notes were off slightly, but as the piano sound is different, it was hard to compare.

Aug 27, 20
Scott Your

6.005 - Elements of Software Construction

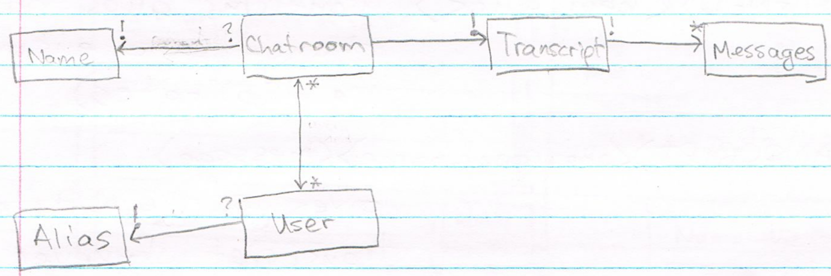
DESIGN PROJECT THREE : GUI Chat

1) Abstract Design

I have opted to go with a chatroom-based IM chat which consists of chatrooms, which can be joined by users. Chatrooms persistently store their conversations and may be created, but cannot be deleted.

Users choose aliases which also appear listed on the interface. A user can only have one alias at a time, but can switch as desired. Inside the chat, all messages are prependended with the alias of the user, but are not altered if the alias later changes (consistent with the idea of logging in as a new user).

Object Model Description:



2) Protocol Design

I've opted to go with a plaintext protocol which has the following options:

Client-to-Server:

<CREATE><chatroom_name> - Initializes and joins a chatroom with the specified name.

<JOIN><chatroom_name> Attempts to join the specified chatroom

<LEAVE><chatroom.name> Leaves the specified chatroom, unsubscribing from updates.

<MSG><chatroom.name><message> Post the specified message to the given chatroom.

<ALIAS><new_alias> Requests your alias to a new name.

Server-to-Client:

<INVITED><chatroom.name> Confirms you have joined a chatroom of the specified name. Needed in case you create a chatroom which already exists (thus prompting _#)

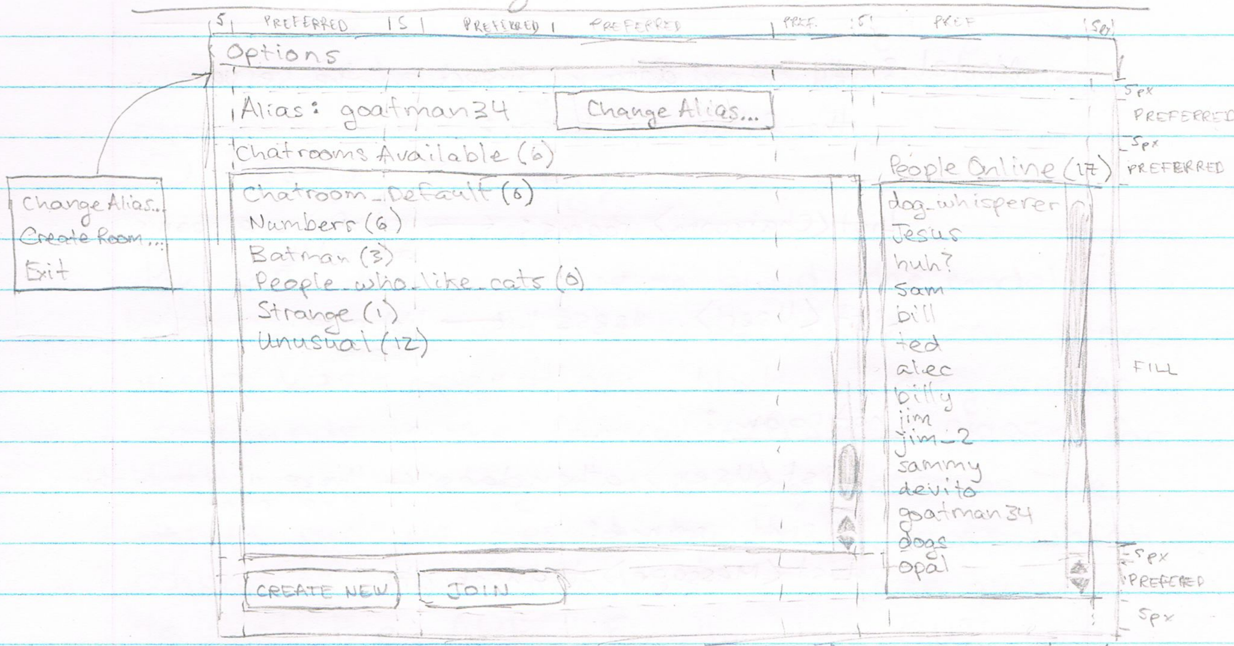
<ASSIGNED><new_alias> Confirms alias name change (and ensures no duplicates)

<TRANSCRIPT><chatroom.name><message>
Gives you the updated transcript whenever a new message is added

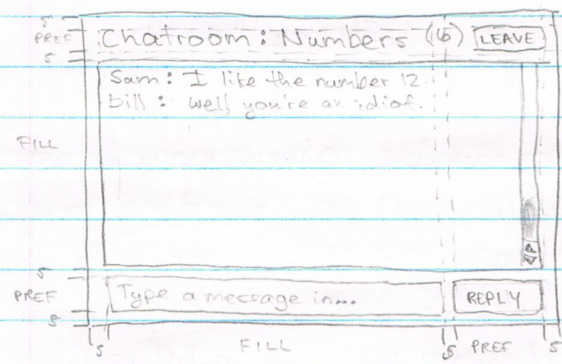
<ROOMLIST><rooms> Updated list of rooms and their # of members (sent whenever a room is created, left or joined)

<USERLIST><aliases> Updated list of ~~users~~ aliases. Sent whenever a person logs in, leaves the program or changes his/her alias.

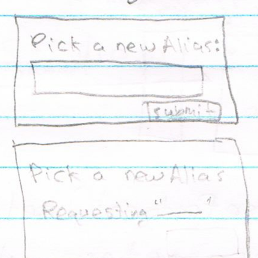
3) Interface Design (UI)

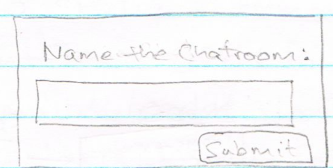


Above: Main window. Chatrooms are a tree, double click to join (or select and pick 'join')

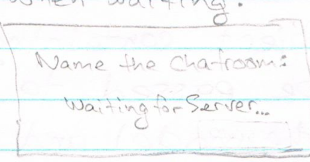


Left: Chatroom Pane. Users are anonymous, so only # of attendees is posted. Closing the window or hitting 'LEAVE' leaves the chatroom. Hitting 'ENTER' or 'REPLY' sends a new msg to the chatroom.





Dialog for creating a new chatroom (exits upon receiving an <INVITED> message from the server, displays (below) when waiting:



Hilroy

4) Code Design

Model: All shared data is stored on the server, in the following form:

List<Chatroom> rooms; ← list of all chatrooms available

List<User> users; ← list of all connected users.

Chatroom:

List<User> attendees; ← those users in this chatroom
String name;
List<Message> transcript;

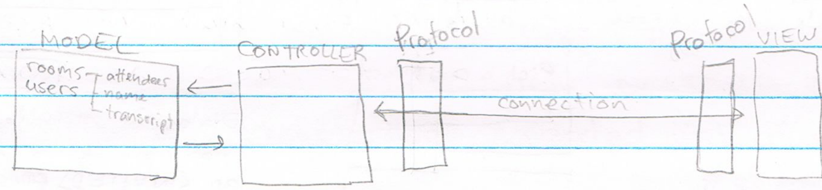
User:

String alias;

Message:

String aliasUsed;
String text;

Controller: The server can be given new protocol messages as well as sending messages to clients.



The command parseProtocol() returns a Protocol object containing parameters which can then be passed to the execute() function, updating the model and submitting any further external messages. All connections and sockets are maintained by the controller.

View: The view only interprets Protocol objects as updates to the interface. Everything else is done by the Controller/Protocol objects.

5) Testing Strategy

My plan for testing is to build the model first, and use unit tests to ensure consistency across basic operations. Next, building server commands in the Controller/Protocol classes for both manipulating and parsing messages. The server can be tested by 'fake' clients which consist of new threads of message queues to test the API. Finally, the GUI will be constructed and hand-tested for errors using the existing API.

Steps:

1) Model → Unit Tests for Basic Operations
2) Controller → Unit Test Commands
3) Protocol → Test Parse → Command Syntax
4) Simulated Client Tests (check for preserved invariants via multithreaded, simulated clients)
5) View → Final Hand-Testing for GUI and Network

Hilroy

6) Reflections

Most of my design changes/blunders were a result of not properly understanding the tools as they should be used, however, other than a frustrating set of errors caused by neglecting to use the invokeLater() command for View actions the project progressed as anticipated. Here are some changes I would incorporate if I wanted to build a more sophisticated IM program:

1) Partial updates. For simplicity I had the server resend all data with any updates, which, in retrospect was very inefficient. Partial updates are more difficult, however.
2) Permanent Storage. Right now, the server deletes all state on close, it would have been nice to have permanent state (by storing the model via SQL, for example).

Overall the project went well and gave me a lot of confidence for building GUI and networking applications in the future.