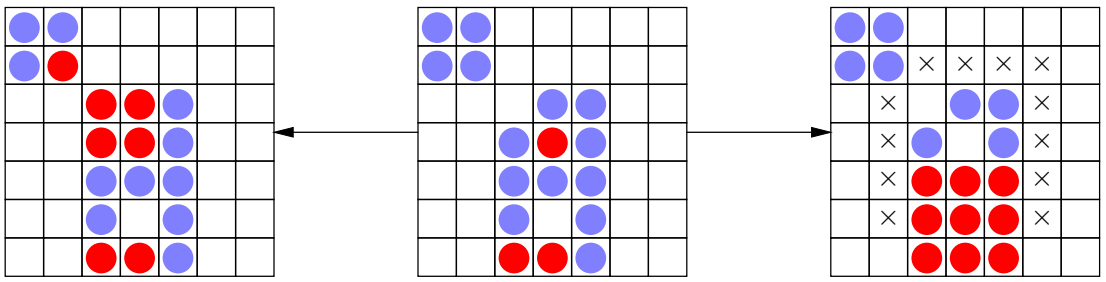
**CS61B, Fall 2008** **Project #3: Ataxx*** **P. N. Hilfinger**

**Due:** Friday, 5 December 2008 at 2400

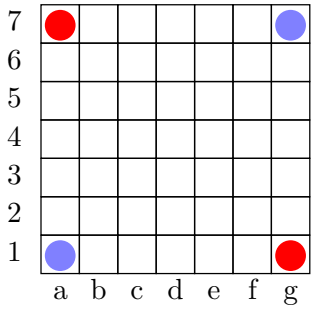
# 1 Background

Ataxx is a two-person game played with red and blue pieces on a 7-by-7 board As illustrated below, there are two possible kinds of move: you can *extend* from a piece of your own color by laying down a new piece of your color in an empty square next to that existing piece (horizontally, vertically, or diagonally), or you can *jump:* move a piece of your own color to an empty, non-adjacent square that is no more than two rows and no more than two columns distant. In either case, all opposing pieces that are next to the previously empty destination square are replaced by pieces of your color. Here is an example of a starting position (in the middle) and two possible moves with the same piece (if you are reading a black-and-white printed copy, the blue pieces will appear to be slightly lighter than the red pieces). Squares marked × show red's other possible jumps with that piece:



*After red move to adjacent square* (`d4-c5`) *Starting position* *After red jump* (`d4-d2`)

At the beginning of the game, we start with pieces in all four corners:



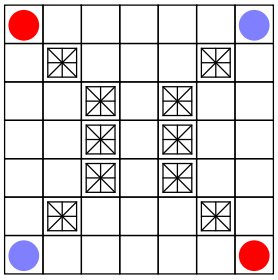

The red player goes first, and play alternates until no more moves are possible, or until there have been 45 jumps with no intervening extends. You are allowed to skip a move only if you have at least one piece, but no legal move. The winner is the player who has the most

*Updated 12/05/2008.

pieces at the end of the game (thus, you don't automatically lose just because you can't move, unless you have no pieces left). It is possible also to tie.

### 1.1 Blocks

To make things even more interesting, you can place a set of *blocks* symmetrically about the center of the board before playing. These are "pre-filled" squares that may never be moved to (the blocks themselves never move). In the illustration below, the left figure is an example of a starting configuration with 10 blocks. The pattern of blocks in the $4 \times 4$ square in the upper-left is always reflected across the middle row and the middle column. Therefore, to describe the configuration of blocks being used, we need merely indicate which squares in the upper-left $4 \times 4$ portion of the board have blocks in them. To do so, we denote these squares with the digits 1–6 and the letters a–h as shown in the illustration on right below. Thus, the configuration on the left would be described "3gh". Redundant letters have no additional effect.



Initial configuration 3gh

| | | | | | | |
|---|---|---|---|---|---|---|
| | d | e | 4 | | | |
| a | g | f | 5 | | | |
| b | c | h | 6 | | | |
| 1 | 2 | 3 | | | | |
| | | | | | | |
| | | | | | | |
| | | | | | | |

Designations of possible block positions

### 1.2 Notation

We'll denote columns with letters a–g from the left and rows with numerals 1–7 from the bottom, as shown in the illustration of the initial position on the previous page. An ordinary move consists of two positions separated by a hyphen in the format "$c_0r_0$–$c_1r_1$" (e.g., `g1-f2`). The first position gives a piece owned by the current player, and the second gives an empty square to which the piece jumps or extends. A single hyphen (surrounded by whitespace) denotes a case where one player must skip a move.

## 2 Textual Input Language

From your program's point of view, we'll refer to the two players as the *Player* and the *Opponent.* The Player is either the person using the program (the *User*) or possibly an automated player (we'll call it an *AI* for short). The Opponent is either an *AI* or a remote opponent (i.e., some other Ataxx program). In any given game, either of these may play either red or blue. The User can talk to the program using either a textual interface, described in this section, or (for extra credit) a GUI.

At any given time, your program has one of three *modes* (sources of moves), and one of three *states* of the game. Your program can either be in *local mode,* meaning that the Opponent is an AI, *host mode,* meaning that the Opponent is a remote program and the User's program is in charge of the game (choosing the color, making any set-up moves, and setting any blocks), or *client mode,* indicating again that the Opponent is a remote program, but that the remote program serves as the host. The game can be in *set-up state,* during which the User (or, in client mode, the Opponent) can set parameters and enter an initial position on the game board, *playing state,* where players are entering moves and the game is not yet won, and *finished state,* where one or the other player has won and no more moves are possible. Initially, the program is in local mode and set-up state.

Your program should respond to the following textual commands (you may add others). There is one command per line, but otherwise, whitespace may precede and follow command names and operands freely. Empty lines have no effect, and everything from a '`#`' character to the end of a line is ignored as a comment.

**Commands to begin and end a game.**

**clear** Abandons the current game (if one is in progress), clears the board to its initial configuration, and places the program in the set-up state and local mode, with the Player being the User. Abandoning a game implies that you resign. Valid in any state.

`start` Valid only in set-up state. Enters playing state, taking moves alternately from the Player and Opponent according to their color and the current move number. If there have been moves made during the set-up state, then play picks up at the point where these moves leave off (so, for example, if the Player is red and there was one set-up move made before '`start`', then the Opponent will move first). In the (unusual) case where the set-up moves have already won the game, `start` puts the program in finished state.

`quit` Abandons any current game (as for `clear`) and exits the program. Valid in any state.

**Set-up.** The following commands are valid only in set-up mode. They set various game parameters prior to the start of play.

`color` *C* Sets the Player's color to *C*, which may be '`red`' or '`blue`'. By default the Player plays red.

`auto` Sets up the program so that the Player is an AI. Thus, in local mode, '`auto`' causes both the Player and Opponent to be AIs, so that the `start` command causes the machine to play a game against itself. Initially and after a `clear` command, the Player is the User.

`blocks` *config* Sets up blocks, as described in §1.1, which gives the syntax for *config.* It is an error for *config* to be empty. It is also an error to place a block on a previously set up piece. In case of any errors, the command has no effect.

`seed` *N* *If* your program's AIs use pseudo-random numbers to choose moves, this command sets the random seed to *N* (a long integer). This command has no effect if there is no random component to your automated players (or if you don't use them in a particular game). It doesn't matter exactly how you use *N* as long as your automated player behaves identically in response to any given sequence of moves from its opponent each time it is seeded with *N*. In the absence of a `seed` command, do what you want to seed your generator.

**Remote play.** The following two commands are valid only in set-up state. They cause the Opponent to become a remote player, whose moves come from another execution of the program when play starts (see §4).

`host` *ID* Here, *ID* is any sequence of letters, underscores, and digits. The program waits for an opponent to join (see the '`join`' command). The program is now in host mode after this command.

`join` *ID*`@`*hostname* There may not be any whitespace in "*ID*@*hostname*." First, clear the board to its initial state, but do not change whether the Player is an AI or the User. There must be a program running on the machine *hostname* (a name like `nova.cs.berkeley`, or, for another program running on the same machine, `localhost`); its user must have entered the `host` *ID* command; and nobody else can have joined the same game. The program is in client mode after this command.

After two players have entered these commands (one executing '`host`' and then the other executing a matchine '`join`'), they remain in set-up state. From this point until it enters playing state, or until the host terminates its connection, the client program executes commands it receives from the host rather than from the User. The host program will first send it the necessary '`color`' command, '`blocks`' command, and set-up moves so that both the client and host have the same board and so that the host's Player has one color and the client's Player has the other. When the host finally sends a `start` command, the client program will alternate reading commands from its Player and from the (now remote) Opponent, as usual.

**Entering moves.** One can enter moves either in set-up state or (when the Player is the User) in playing state. In set-up state, moves serve to manually set up a position on the playing board. The first and then every other move is for the red player, the second and then every other is for blue, and the normal legality rules apply to all moves. In playing state, the Player enters every other move, depending on color, and the Opponent supplies the rest. See §1.2 for move syntax.

Your program must reject illegal moves from the Player (they have no effect on the board; the program should tell the user that there is an error and request another move; an AI should not make illegal moves). Your program should never send an erroneous move to a remote Opponent. If it receives an erroneous move from a remote Opponent, this is a fatal error. Recover by going to finished state, terminating any connection, and announcing that the Player has won.

**Miscellaneous commands.** The following commands are valid in any state.

`help` Print a brief summary of the commands. ¡

`dump` This command is especially for testing and debugging. It prints the board out in *exactly* the following format:

```
===
  r - - - - - b
  - - - - - - -
  - - X - X - -
  - - - - - - -
  - - X - X - -
  - - - - - - -
  b - - - - - r
===
```

Here, '`-`' indicates an empty square, '`r`' indicates a red square, '`b`' indicates a blue square, and '`X`' indicates a block. Don't use the two '`===`' markers anywhere else in your output. This gives the autograder a way to determine the state of your game board at any point. It does not change any of the state of the program.

`load` *file* Reads the given *file* and in effect substitutes its contents for the `load` command itself.

## 3 Output

This time, you can prompt however you want, and print out whatever user-friendly output you wish (other than debugging output or Java exception tracebacks, that is). For example, you will probably want to print the game board out after each move is complete (this is distinct from printing the board in the special format required by '`dump`'). When users enter erroneous input, you should print an error message, and the input should have no effect (and in particular, the user should be able to continue entering commands after an error).

When either player enters a winning move, the program should print a line saying either `"Red wins."`, `"Blue wins."`, or `"Draw."` as appropriate. Use exactly those phrases, alone on their line. At that point, each program goes into finished state (maintaining the final state of the board so that the User may examine it). Don't use these phrases in any other situation. In particular, if a player forfeits, use some other phrase to announce it (like `"Red forfeits."` or `"Blue wins by forfeit."`)

## 4 Communicating with a Remote Program

Java supplies a Remote Method Invocation (RMI) package that allows two separate programs (possibly on different machines) to communicate with each other by calling each other's methods. In effect, one program can have pointers (called *remote pointers*) to objects in the other program. We have developed our own packages that allow you to make use of this facility.

You can communicate with a remote job by means of a "mailbox" abstraction that we supply in the form of classes in the package `ucb.util.mailbox`. Take a look at the interface `Mailbox` in that package (in the on-line documentation). The idea is that a mailbox is simply a kind of queue. Its methods allow you to *deposit* messages into it, and to wait for and *receive* messages that have been deposited into it, in the order they were deposited. You can do this even if the mailbox is on another machine. The class `QueuedMailbox` is probably the only implementation of `Mailbox` you'll need.

To talk to a remote program, you will employ two mailboxes: one to send it messages and one to receive messages from it. Both mailboxes will reside in the host program (the one that issues the `host` command). The messages they send each other will be a subset of the commands: the host or the client may send `"stop"` or moves; the host alone may additionally send `"start"`, `"color"`, `"blocks"`; the client alone may send `"ready"`. The format is less free: a single space between operands, and no leading or trailing whitespace.

The tricky part is getting pointers to the mailboxes from one program to the other. For this purpose, we provide another useful type: `ucb.util.SimpleObjectRegistry`. A registry is like a Map, in that it allows one to associate names with values (object references). Suppose that the host player has entered '`host Foo`' and is running on machine $M$. The host program creates a `SimpleObjectRegistry` and stores two `Mailboxes` in it named `"Foo.IN"` and `"Foo.OUT"` using the `rebind` method. When the joining (client) program executes `join FOO@`$M$, it retrieves these `Mailboxes` using (for example)

```
(Mailbox<String>) SimpleObjectRegistry.findObject("Foo.IN","M")
```

The client program sends messages to the host by depositing them into the mailbox `Foo.IN`, and reads messages from the host out of `Foo.OUT`. The roles of the two mailboxes are reversed in the host program, of course.

Once the client program has fetched remote pointers to `Foo.IN` and `Foo.OUT`, it informs the host program that it is ready to begin a game by sending the message `"ready"` in `Foo.IN`. It then executes commands sent from the host, reading them from `Foo.OUT`, and executes them as if they came from the Player. On receiving this `"ready"` command, the host program's `host` command finishes, and the host's Player continues entering set-up commands. When the host's Player enters `start`, the host program then uses `Foo.OUT` to send `"color` $C$`"`, where $C$ is the *client's* color, and, if necessary, the `blocks` command, any set-up moves that the host's Player had entered prior to typing '`start`', and (of course), the '`start`' command itself. At that point, play alternates between the two Players (host and client) as usual. The host sends moves from its Player (when its his turn, that is) and reads moves from the client's Player from `Foo.IN`. For its part, the client reads and executes all these messages from `Foo.OUT`, and upon seeing `"start"`, begins sending moves through `Foo.IN`, and receiving responses from `Foo.OUT`. When the game ends, whoever sent the last (winning) move should then wait for the other side to send `"stop"`, acknowledging receipt of the last move and ending the game. In the unusual case where the set-up moves have already won the game, the "winning move" is the '`start`' command from the host. Either side can also send `"stop"` (in turn) to abandon (and thus resign) the current game (your program would send this, for example, if your Player executed '`clear`' or '`quit`' to tell the remote program that it had abandoned the game).

Any other sequence of messages is illegal. Your program can recover when it detects an illegal message sequence by going to finished state (and presumably printing an error message).

The messages are all strings with no leading or trailing whitespace and with single blanks used to separate commands and operands (where needed). Only the messages '`ready`' (from the client only), '`blocks`', '`color`', '`stop`', and move commands (such as `"c1-b1"` or `"-"`) are allowed.

Upon sending or receiving its last message for a game to one of these mailboxes, the host program should apply its `.close` method to make sure that the message has been received and to shut down the mailbox. Also, the host should close its repository when it receives the first message from the client.

**Annoying technical glitch.** When the program that creates a remote object (a mailbox in this case) terminates, the object is destroyed, and attempts to call its methods (including `.close` get `RemoteExceptions`. If you get such an exception, it probably means that the other program has terminated. Be prepared to receive such exceptions (that is, don't simply surround everything with

```
try {
   ...
} catch (RemoteException e) { }
```

and effectively ignore the exception.) If you're in the middle of a game, interpret such an exception as a `"stop"` message. If you receive one of these at the end of the game (when closing the mailbox), then you can ignore it.

## 5 Running Your Program

Your job is to write a program to play Ataxx. Appropriately enough, we'll call the program `ataxx`. Your AIs can play arbitrarily stupidly, as long as they make only legal moves. However, there will be a tournament at the end. For extra credit, you can provide a GUI (after all, what computer board game is complete without a GUI?).

To run your program, the User types

```
java ataxx [ --display ]
```

The optional parameter `--display` indicates that the User will communicate through a graphical interface (GUI). Your program must work without this option (that's how we test it). If you choose not to implement a GUI, make sure that your program terminates with an error exit code other than 0 when given the `--display` parameter. Otherwise, your program should exit with exit code 0. Even if someone entered invalid commands during a session, your program should exit normally (and, of course, print error messages in response to the errors).

## 6 Your Task

The staff directory will contain skeleton files for this project in `proj3`.

Please read *General Guidelines for Programming Projects.*

Be sure to include tests of your program (that is part of the grade). The makefile we provide has a convenient target for running such tests. Our skeleton directory contains a couple of trivial tests, but *these do not constitute an adequate set of tests!* Make up your tests ahead of time and update your makefile to run them. To help with testing and debugging, we will provide our own version of the program, so that you can test your program against ours (we'll be on the lookout for illegal moves). More details will follow.

The input to your program will come from fallible humans. Therefore, part of the problem is dealing gracefully with errors. When the user makes an illegal move, tell him and just give him another chance. We don't care about the message format you use to do this.

Be sure to include documentation. This consists of a user's manual explaining how to use your program, and a brief internals document describing overall program structure, and any important data structures and algorithms you use (especially, how does your machine player choose its moves?).

Our testing of your projects (but not our grading!) will be automated. The testing program will be finicky, so be sure that

```
java ataxxgame.Testing
```

runs your tests.

## 7 Advice

At first glance, it might seem that with all the options available, the program would have to be a mass of **if** statements covering all possible cases. But with proper use of abstraction, this does not have to be.

We've included documentation for the `ucb.util` and `ucb.util.mailbox` packages in the on-line materials. You will also want to read Chapter 11 of *A Java Reference* and Chapter 10 of *Data Structures (Into Java)* (on concurrency). If you need random numbers, take a look at `java.util.Random` and Chapter 11 of *Data Structures (Into Java).*

I suggest working first on the classes `Board`, which is supposed to represent the game board, and `Command`, which houses some command-parsing code. Next, implement the human player. If you now get the remote game part working (it's not really as hard as it sounds) you can run two versions of your program and play against yourself, checking that the board works properly. Then you can tackle writing a machine player. Start with something really simple (perhaps choosing a legal move at random) and introduce strategy only when you get everything working properly.