CS61B, Spring 2006 **Project #3: Jumping Cubes** **P. N. Hilfinger**

Due: 5 May 2006

1 Background

The KJumpingCube game[1] is a simple two-person board game. It is a pure strategy game, involving no element of chance. For this final project, you are to implement our version of this game, which we'll call `jump61b`, allowing a user to play against a computer or against a remote opponent (someone running another JumpingCube program, possibly on a different machine). The basic interface is textual, as before, but for extra credit, you can produce a GUI interface for the game.

To allow playing against remote versions of the game (versions written by other people that is), there is a simple message-passing protocol for sending moves to another program. Naturally, your program will have to conform to this protocol exactly; otherwise chaos will ensue. We will use this protocol to test your program, in fact.

2 Rules of Jump61b

The game board consists of an $N \times N$ array of squares, where $N > 1$. At any time, each square may have one of three colors: red, blue, or white (neutral), and some number of *spots* (as on dice). Initially, all squares are white and have one spot.

For purposes of naming squares, we'll use the following notation: $r : c$ refers to the square at row r and column c, where $1 \leq r, c \leq N$. Rows are numbered from top to bottom (top row is row 1) and columns are numbered from the left.

The *neighbors* of a square are the horizontally and vertically adjacent squares (diagonally adjacent squares are not neighbors). We say that a square is *overfull* if it contains more spots than it has neighbors. Thus, the four corner squares are overfull when they have more than two spots; other squares on the edge are overfull with more than three spots; and all others are overfull with more than four spots.

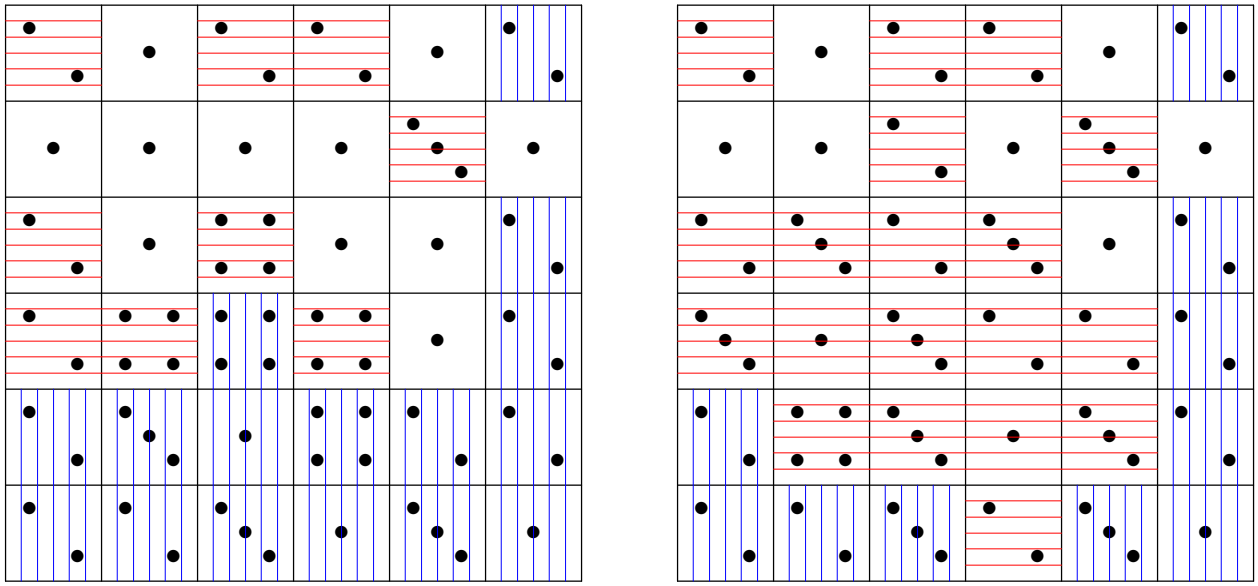
There are two players, whom we'll call Red and Blue. The players each move in turn, with Red going first. A move consists of adding one spot on any square that does not have the opponent's color (so Red may add a spot to either a red or white square). A spot placed on a white square colors that square with the player's color. After the player has moved, we repeat the following process until no square is overfull or all squares are the same color:

1. Pick an overfull square.
2. For each neighbor of the overfull square, move one spot out of the square and into the neighbor.
3. Give each of these neighboring squares the player's color (if they don't have it already).

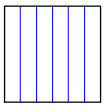
[1] Distributed under the GNU Public License as part of the KDE project. Copyright 1999, 2000 by Matthias Kiefer. It was inspired by an old game on the Commodore 64.

The order in which this happens, as it turns out, does not usually matter—that is, the end result will be the same regardless of which overfull square's spots are removed first, with the exception that the winning position might differ. A player wins when all squares are the player's color.

For example, given the board on the left ($N = 6$), if Red adds a spot to square 3:3, we get the board on the right after all the spots stop jumping.



Legend:

Red square:  Blue square: 

The rules hold that the game is over as soon as one player's color covers the board. This is a slightly subtle point: it is easy to set up situations where the procedure given above for dealing with overfull squares loops infinitely, swapping spots around in an endless cycle, unless one is careful to stop when a winning position appears.

3 Textual Input Language

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 part of Player 2 is taken by the program itself, *host mode,* meaning that the part of Player 2 is taken by a remote program and your program is in charge of the game (decides colors and board size), or *client mode,* indicating again that Player 2 is taken by a remote program, but that the remote program serves as the host. The game can be in *set-up state,* during which any moves entered are only intended to set up 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 players 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.

We'll use the term *Player 1* to indicate the user typing at the console, and *Player 2* to indicate the other player.

Commands to begin and end a game.

clear Abandons the current game (if one is in progress), clears the board to its initial configuration (all squares neutral), and places the program in the set-up state. Abandoning a game implies that you resign.

`start` If in playing or finished state, first executes a '`clear`' command. Then enters the playing state, taking moves alternately from Player 1 and Player 2 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 Player 1 is red and there was one set-up move made before '`start`', then Player 2 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 and exits the program.

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

`game` *size color* Sets the size of the board to *size* squares on a side, and clears the board to its initial configuration. It also causes Player 1 to play the given *color,* which may be '`red`' or '`blue`'. By default (until the first '`game`' command), the board is 6×6 and Player 1 plays red.

`auto` Sets up the program so that Player 1's moves come from an automated player rather than the user at the console. Thus, in local mode, '`auto`' causes the machine to play a game against itself.

`manual` Counters the effect of '`auto`', causing Player 1's moves to come from the terminal. This is the default.

Remote play. The following two commands are valid only in set-up state. They cause Player 2 to become a remote player, whose moves come from another execution of the program when play starts (see §5).

`host` *ID* Here, *ID* is any sequence of letters, underscores, and digits. First, this command clears the board to its initial state. Then it sets up a game against a remote opponent. It waits for an opponent to join (see the '`join`' command). The program is in host mode after this command.

`join` *ID*`@`*hostname* There may not be any whitespace in "*ID*@*hostname*." 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, they remain in set-up state. The client program executes commands from the host rather than the terminal until the host sends '`start`'. At that point, your program will alternate reading commands from the terminal (or from an automated player if '`auto`' is in effect) for Player 1's moves and from the remote program for Player 2's. When the game is over (or if the remote Player 2 quits), the program returns to local mode (disconnecting from the remote program).

Entering moves. One can enter moves either in set-up state or (when '`manual`' is in effect), in playing state. In set-up state, moves serve to manually set up a position on the playing board. In either case, 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.

R`:`*C* Adds a spot to the square at row *R*, column *C*, where *R* and *C* are integers in the range 1 to the current board size. After the spot is added, spots are redistributed as indicated in the rules above. There may not be whitespace embedded in this command. As indicated in the rules, spots alternate between going on red or neutral squares, and on blue or neutral squares. Illegal moves must be rejected (they have no effect on the board; the program should tell the user that there is an error and request another move).

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:

```
===
    2r -- 2r 2r -- 2b
    -- -- 2r -- 3r --
    2r 3r 2r 3r -- 2b
    3r 1r 3r 2r 2r 2b
    2b 4r 3r 1r 3r 2b
    2b 2b 3b 2r 3b 1b
===
```

Here, '`--`' indicates a neutral square, '*N*`r`' indicates a red square with *N* spots, and '*N*`b`' indicates a blue square with *N* spots. 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.

`seed` *N* *If* your program's automated players (used in local mode and with the '`auto`') 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, you do what you want to seed your generator.

4 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."` or `"Blue wins."` as appropriate. Use exactly those phrases.

5 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 `"clear"`, `"quit"`, or moves "$R : C$"; the host only may additionally send `"game"` and `"start"`. However, 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: `java.util.SimpleObjectRegistry`. A registry is like a Map, in that it allows one to associate names with values (object references). Suppose that the host program has executed '`host Foo`' and is running on machine *M*. The host player eventually issues a `start` command. The host program creates a `SimpleObjectRegistry` and stores two `Mailboxes` in it named `"Foo.IN"` and `"Foo.OUT"` using the `rebind` method. The

joining (client) program executes `join FOO.`M, and when it executes a `start` command, 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 has fetched remote pointers to `Foo.IN` and `Foo.OUT`, it informs the host that it is ready to begin a game by sending the message `"clear"` in `Foo.IN`. It then executes commands sent from the host, reading them from `Foo.OUT`. The host then uses `Foo.OUT` to send `"game` N C`"`, where N is the board size and C is the *client's* color, and any set-up moves that the host's Player 1 enters prior to typing '`start`'. If the host's Player 1 enters another '`game`' command, the host sends that command along as well, again with the color reversed. Finally, the host sends the `"start"` message when its Player 1 enters that command. The host then sends moves from its Player 1 (when its his turn, that is) and reads moves from the client's Player 1 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 `"quit"`, acknowledging receipt of the last move and ending the game (receiving a `"quit"` message does *not* your program should terminate, merely that it should return to local mode). 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 `quit` to abandon the current game.

Any other sequence of messages is illegal. Your program can recover when it detects an illegal message sequence by returning to set-up state in local mode (and presumably printing an error message).

Upon sending your last message to one of these mailboxes, 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 `"quit"` message. If you receive one of these at the end of the game (when closing the mailbox), then you can ignore it.

6 Advice

Start as soon as the skeleton files are available! The on-line version of this handout has additional advice, information, and examples. We'll post additional notes on the lab page as well, with more advice and possibly corrections as needed.