6.005 Elements of Software Construction | Fall 2011
Problem Set 6: Multiplayer Minesweeper

The purpose of this problem set is to explore multithreaded programming with a shared mutable datatype, which you should protect using synchronization.

You have substantial design freedom on this problem set. However, in order for your solution to be graded, your solution must not change the name, method signature, class name, package name, or specification of the following:

- minesweeper.server.MinesweeperServer.main()

Overview

You will start with some minimal server code and implement a server and thread-safe data structure for playing a multiplayer variant of the classic computer game "Minesweeper"

You can review the traditional/single-player minesweeper concept / rules here: http://en.wikipedia.org/wiki/Minesweeper_(video_game)

You can try playing traditional/single-player minesweeper here: http://www.chezpoor.com/minesweeper/minesweeper.html [1]

the final product will consist of a server and no client; it should be fully playable using telnet.

Notes

We will refer to the board the board as an NxN grid where each square has a state which can be 'flagged', 'dug', or 'untouched', and each square either has a bomb or does not have a bomb.

Our variant works very similarly to real minesweeper but with multiple players for one board. the main functional difference is that when one player blows up a bomb in single player, they just lose. when one player blows up a bomb in our version, they still lose, [i.e. server ends their connection] but the other players may continue playing. The square where the bomb was blown up is now a dug square with no bomb. (The player who lost may also reconnect to the same game via telnet.)

Note that there are some tricky cases of user-level concurrency: as a notable example, user A has just modified the game state (i.e. by digging in one or more squares) such that square i,j obviously has a bomb. Meanwhile, user B has not observed the board state since this update has taken place, so user B goes ahead and digs in square i, j. Your program should allow the user to dig in that square-- a user of Multiplayer Minesweeper must accept this kind of risk.

We are not specifically defining, or asking you to implement, any kind of "win condition" for the game.

In our version of minesweeper, the board is always square.

Protocol and Specification

You must implement the following protocol for communication between the user and the server, and associated specification.

Grammar for messages from the user to the server:

```
        User-to-Server Minesweeper Message Protocol
          MESSAGE     :== ( LOOK | DIG | FLAG | DEFLAG | HELP_REQ | BYE ) NEWLINE
          LOOK        :== "look"
          DIG         :== "dig" SPACE X SPACE Y
          FLAG        :== "flag" SPACE X SPACE Y
          DEFLAG      :== "deflag" SPACE X SPACE Y
          HELP_REQ    :== "help"
          BYE         :== "bye"
          NEWLINE     :== "\n"
          X           :== INT
          Y           :== INT
          SPACE       :== " "
          INT         :== [0-9]+
```

Server spec for dealing with user input:

LOOK message:

Returns a BOARD message, a string representation of the board’s state. Does not mutate anything on the server. See SERVER->CLIENT protocol for the grammar of the BOARD message.

DIG message:

1. If either x or y is less than 0 or greater than board size, or square x,y is not in the 'untouched' state, do nothing and return a BOARD message.
2. If square x,y's state is 'untouched', change square x,y's state to 'dug'.
3. If square x,y has a bomb, change it so it has no bomb and send a BOOM message to the user (see SERVER-USER protocol). Then, if the DEBUG flag is not set to 'true' (see Question 4), terminate the user's connection.
4. For any DIG message where a BOOM message is not returned, return a BOARD message.
5. If this operation results in a 'dug' square whose neighbors all have no bomb, perform the DIG operation on all of those squares.

FLAG message:

1. If x and y are both greater than or equal to 0, and less than board size, and square x,y is in the 'untouched' state, change it to be in the 'flagged' state.
2. Otherwise, do not mutate any state on the server.
3. For any FLAG message, return a BOARD message.

DEFLAG message:

1. If x and y are both greater than or equal to 0, and less than board size, and square x,y is in the 'flagged' state, change it to be in the 'untouched' state.
2. Otherwise, do not mutate any state on the server.
3. For any DEFLAG message, return a BOARD message.

HELP_REQ message:

Returns a HELP message as defined in the SERVER-USER protocol. Does not mutate anything on the server.

BYE message:

Terminate the connection with this client.

To clarify, for any message which matches the grammar, other than a BYE message, we should always be returning either a BOARD message, a BOOM message, or a HELP message.

For any server input which does not match the USER->SERVER grammar, do nothing.

Grammar for messages from the server to the user:

```
MESSAGE :== BOARD | BOOM | HELP | HELLO
BOARD :== LINE+
LINE :== (SQUARE SPACE)* SQUARE NEWLINE
SQUARE :== “-” | “F” | COUNT | SPACE
SPACE :== “ “
NEWLINE :== “\n”
COUNT :== [1-8]
BOOM :== "BOOM!" NEWLINE
HELP :== [^NewLine]+ NEWLINE
HELLO :== "Welcome to Minesweeper.  " N " people are playing including you.  Type 'help' for
help." NEWLINE
N :== INT
INT :== [0-9]+
```

The **BOARD** message, as the grammar indicates, consists of a series of newline-separated rows of space-separated chars, thereby giving a grid representation of the board's state with exactly one char for each square. The mapping of chars is as follows:

- "-" for squares with state "untouched".
- "F" for squares with state "flagged".
- " " for squares with state "dug" and 0 neighbors who have a bomb.
- integer COUNT in range [1-8] for squares with state "dug" and COUNT neighbors who have a bomb.

Notice that in this representation we reveal every square's state of "untouched", "flagged", or "dug", and we indirectly reveal limited information about whether some squares have bombs.

The **HELP** message should print out a message which indicates all the commands the user can send to the server (the exact design of the message is up to you.)

As the grammar indicates, the **HELLO** message includes N which is the number of users currently connected to the server. This message should be send to the user only once, immediately after the server connects to the user.

Problem 1: Setting up a Server to Deal with Multiple Clients

a. [15 points] We have provided you with a single-thread server which can accept connections with one client at a time, and which includes code to parse the input according to the client-server protocol above. Modify the server so it can maintain multiple client connections simultaneously. Each client connection should be maintained by its own thread. You may wish to add another class to do this. You may continue to do nothing with the parsed user input at this time.

Problem 2: Implementing a Data Structure for Minesweeper

a. [30 points] Specify, implement, and test the minesweeper board data structure (as a Java type, without using sockets or threads). You are encouraged to add additional classes beyond the Board class that we have provided for you.

Problem 3: Making your Data Structure Thread Safe

a. [15 points] Make the minesweeper board threadsafe using synchronization (again, just using Java method calls, not sockets).

b. [5 points] Near the top of your Board.java source file, include a substantial comment with an argument about why your board is thread-safe.

Problem 4: Setting up the Server to take Command Line Arguments

We will now add a few command line arguments to MinesweeperServer. Here is the protocol for the arguments:

```
ARGS :== DEBUG SPACE ( SIZE | FILE )?
DEBUG :== "true" | "false"
SIZE :== SIZE_FLAG SPACE X
SIZE_FLAG :== "-s"
X :== INT
FILE :== FILE_FLAG SPACE PATH
FILE_FLAG :== "-f"
PATH :== .+
INT :== [0-9]+
SPACE :== " "
```

To fulfill the server's specification, we will need the server to have an instance of our Board data structure.

For the **DEBUG** argument, the server should set a boolean DEBUG flag with the corresponding value (this will be used in Question 5.)

For a **SIZE** argument: if X > 0, the server's Board instance should be randomly generated and should have size equal to X by X. To randomly generate your board, you should assign each square to have a bomb with probability .25; else no bomb. All squares' states should be set to 'untouched'.

For a **FILE** argument: If a file exists at the given PATH, read the the corresponding file, and if it is properly formatted, deterministically create the Board instance. The file format for input should be:

```
FILE :== LINE+
LINE :== (VAL SPACE)* VAL NEWLINE
VAL :== 0 | 1
SPACE :== " "
NEWLINE :== "\n"
```

In a properly formatted file matching the FILE grammar, if there are N LINEs, each line must contain N VALs. If the file read is properly formatted, the Board should be instantiated such that square i,j has a bomb iff the j'th VAL in LINE i of the input is 1.

In if neither a SIZE or FILE argument is present, the Board should be randomly generated as in the case of a SIZE argument, but with a size of 10 by 10.

For example, if you were running your server from the command line and the executable was called 'server', some command line arguments might look like:

```
./server true
./server false -s 30
./server true -f ../testBoard
```

a. [10 points] Modify your server so that it parses these command line arguments according to the grammar, and fulfills our specification for the arguments.

Problem 5: Putting it all Together

a. [20 points] Modify your server so that it implements our protocols and specifications, by keeping a shared reference to a single instance of Board. Note that when we send a BOOM message to the user, we should terminate their connection iff the DEBUG flag (set as instructed in problem 4) is set to 'false'.

b. [5 points] Near the top of your MinesweeperServer.java source file, include a substantial comment with an argument about why your server is thread-safe.

[1] You may notice that this implementation does something subtle: it ensures that there's never a bomb where you make your first click of the game. You should not implement this for the assignment. (It would be in conflict with giving the option to pass in pre-designed board, for example.)