Problem Set 2

Bookmark this page

Problem Set 2 Starting Code

Download the starting code for Problem Set 2 here:

ps2.zip

The process for doing this problem set is the same as in Problem Set 1, but here are quick reminders:

- To import the starting code into Eclipse, use File → Import... → General → Existing Projects Into Workspace → Select Archive File, then Browse to find where you downloaded ps2.zip. Make sure `ps2-minesweeper` is checked and click Finish.
- To run JUnit tests, right-click on the `test` folder in Eclipse and choose Run As → JUnit Test.
- This problem set has no `main()` method to run, just tests.
- To run the autograder, right-click on `grader.xml` in Eclipse and choose Run As → Ant Build.
- To view the autograder results, make sure your project is Refreshed, then double-click on `my-grader-report.xml`.
- To submit your problem set, upload `my-submission.zip` to the submission page, which is the last section of this handout, at the end of the section bar.

Problem Set 2: Multiplayer Minesweeper

Overview

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

- You can review the **rules of traditional single-player Minesweeper** on Wikipedia.
- You can also **play traditional single-player Minesweeper** online.

  (You may notice that the implementation in the latter link above 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 a pre-designed board, for example.)

Your final product will consist of a server and no client; it should be fully playable using the `telnet` utility to send text commands directly over a network connection as described below.

Multiplayer Minesweeper server

We will refer to the board as a grid of squares. Each square is either *flagged*, *dug*, or *untouched*. Each square also either contains a bomb, or does not contain a bomb.

Our variant works very similarly to standard Minesweeper, but with multiple players simultaneously playing on a single board. In both versions, players lose when they try to dig an untouched square that happens to contain a bomb. Whereas a standard Minesweeper game would end at this point, in our version, the game keeps going for the other players. In our version, when one player blows up a bomb, they still lose, and the game ends for them (the 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 reconnect to the same game again via `telnet` to start playing again.

Note that there are some tricky cases of user-level concurrency. For example, say 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.

Telnet client

`telnet` is a utility that allows you to make a direct network connection to a listening server and communicate with it via a terminal interface. Before starting this problem set, please ensure that you have `telnet` installed. *nix operating systems (including Mac OS X) should have `telnet` installed by default.

Windows users should first check if telnet is installed by running the command `telnet` on the command line.

- If you do not have `telnet`, you can install it via Control Panel → Programs and Features → Turn Windows features on/off → Telnet client. However, this version of `telnet` may be very hard to use. If it does not show you what you're typing, you will need to turn on the `localecho` option.
- A better alternative is PuTTY: download `putty.exe`. To connect using PuTTY, enter a hostname and port, select Connection type: Raw, and Close window on exit: Never. The last option will prevent the window from disappearing as soon as the server closes its end of the connection.

You can have `telnet` connect to a host and port (for example, `web.mit.edu:80`) from the command line with:

```
telnet web.mit.edu 80
```

Since port 80 is usually used for HTTP, we can now make HTTP requests through `telnet`. If you now type (where ↵ indicates a newline):

```
GET /↵
```

`telnet` should retrieve the HTML for the webpage at web.mit.edu. If you want to connect to your own machine, you can use `localhost` as the hostname and whatever port your server is listening on. With the default port of 4444 in this problem set, you can connect to your running Minesweeper server with:

```
telnet localhost 4444
```

The server may close the network connection at any time, at which point `telnet` will exit. To close a connection from the client, note the first output printed by telnet when it connects:

```
Trying 18.9.22.69...
Connected to web.mit.edu.
Escape character is '^]'.
```

`^]` means `Ctrl-]`, so press that. Then enter `quit` to close the connection and exit telnet.

Protocol and specification

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

The messages in this protocol are described precisely and comprehensively using a pair of grammars. Your server must accept any incoming message that satisfies the user-to-server grammar, react appropriately, and generate only outgoing messages that satisfy the server-to-user grammar.

- Wikipedia description of the **grammar notation: Backus–Naur Form**
- Wikipedia description of **regular expressions**, which are used on the right-hand side of grammar productions

Messages from the user to the server

Formal grammar

```
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" | "\r" "\n"?
X ::= INT
Y ::= INT
SPACE ::= " "
INT ::= "-"? [0-9]+
```

Each message starts with a message type and arguments to the message are separated by a single SPACE character and the message ends with a NEWLINE. The NEWLINE can be a single character `"\n"` or `"\r"` or the two-character sequence `"\r\n"`, the same definition used by `BufferedReader.readLine()`.

The user can send the following messages to the server:

LOOK message

The message type is the word “look” and there are no arguments.

Example:

l	o	o	k	\n

Returns a BOARD message, a string representation of the board’s state. Does not mutate anything on the server. See the section below on messages from the server to the user for the exact required format of the BOARD message.

DIG message

The message is the word “dig” followed by two arguments, the X and Y coordinates. The type and the two arguments are separated by a single SPACE.

Example:

d	i	g		3		1	0	\r	\n

The dig message has the following properties:

1. If either x or y is less than 0, or either x or y is equal to or greater than the 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 contains a bomb, change it so that it contains no bomb and send a BOOM message to the user. Then, if the debug flag is missing (see Question 4), terminate the user’s connection. See again the section below for the exact required format of the BOOM message. Note: When modifying a square from containing a bomb to no longer containing a bomb, make sure that subsequent BOARD messages show updated bomb counts in the adjacent squares. After removing the bomb continue to the next step.
4. If the square x,y has no neighbor squares with bombs, then for each of x,y’s *untouched* neighbor squares, change said square to *dug* and repeat *this step* (not the entire DIG procedure) recursively for said neighbor square unless said neighbor square was already dug before said change.
5. For any DIG message where a BOOM message is not returned, return a BOARD message.

FLAG message

The message type is the word “flag” followed by two arguments the X and Y coordinates. The type and the two arguments are separated by a single SPACE.

Example:

f	l	a	g		1	1		8	\n

The flag message has the following properties:

1. If x and y are both greater than or equal to 0, and less than the 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

The message type is the word “deflag” followed by two arguments the X and Y coordinates. The type and the two arguments are separated by a single SPACE.

Example:

d	e	f	l	a	g		9		9	\n

The flag message has the following properties:

1. If x and y are both greater than or equal to 0, and less than the 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

The message type is the word “help” and there are no arguments.

Example:

h	e	l	p	\r	\n

Returns a HELP message (see below). Does not mutate anything on the server.

BYE message

The message type is the word “bye” and there are no arguments.

Example:

b	y	e	\r	\n

Terminates the connection with this client.

Messages from the server to the user

Formal grammar

```
MESSAGE ::= BOARD | BOOM | HELP | HELLO
BOARD ::= LINE+
LINE ::= (SQUARE SPACE)* SQUARE NEWLINE
SQUARE ::= "-" | "F" | COUNT | SPACE
SPACE ::= " "
NEWLINE ::= "\n" | "\r" "\n"?
COUNT ::= [1-8]
BOOM ::= "BOOM!" NEWLINE
HELP ::= [^\r\n]+ NEWLINE
HELLO ::= "Welcome to Minesweeper. Players: " N " including you. Board: "
          X " columns by " Y " rows. Type 'help' for help." NEWLINE
N ::= INT
X ::= INT
Y ::= INT
INT ::= "-"? [0-9]+
```

There are no message types in the messages sent by the server to the user. **The server sends the HELLO message as soon as it establishes a connection to the user.** After that, for any message it receives that matches the user-to-server message format, other than a BYE message, the server should always return either a BOARD message, a BOOM message, or a HELP message.

For any message from the user which does not match the user-to-server message format, discard the incoming message and send a HELP message as the reply to the user.

The action to take for each different kind of message is as follows:

HELLO message

In this message:

- *N* is the number of users currently connected to the server, and
- the board is *X* × *Y*.

This message should be sent to the user exactly as defined and only once, immediately after the server connects to the user. Again the message should end with a NEWLINE.

Example:

W	e	l	c	o	m	e		t	o		M	i	n	e	s	w	e	e	p
e	r	.		P	l	a	y	e	r	s	:		1	2		i	n	c	l
u	d	i	n	g		y	o	u	.		B	o	a	r	d	:		8	
c	o	l	u	m	n	s		b	y		8		r	o	w	s	.		T
y	p	e		'	h	e	l	p	'		f	o	r		h	e	l	p	.
\r	\n																		

BOARD message

The message has no start type word and consists of a series of newline-separated rows of space-separated characters, thereby giving a grid representation of the board's state with exactly one char for each square. The mapping of characters is as follows:

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

Here is an example board message with 3 rows and 8 columns:

										2		-		-	\r	\n
1		1						1		F		-		-	\n	
F		1						1		-		-		-	\n	

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 or not.

In the printed **BOARD** output, the (x,y) coordinates start at (0,0) in the top-left corner, extend horizontally to the right in the X direction, and vertically downwards in the Y direction. (While different from the standard geometric convention for IV quadrant, this happens to be the protocol specification.) So the following output would represent a flagged square at **(x=1,y=2)** and the rest of the squares being untouched:

```
- - - - - - -
- - - - - - -
- F - - - - -
- - - - - - -
- - - - - - -
- - - - - - -
- - - - - - -
```

In order to conform to the protocol specification, you'll need to preserve this arrangement of cells in board while writing the board messages. If you change this order in either writing the board message or reading the board from a file (Problem 4) your implementation does not satisfy the spec.

BOOM message

The message is the word *BOOM!* followed by a NEWLINE.

Example:

B	O	O	M	!	\r	\n

The last message the client will receive from the server before it gets disconnected!

HELP message

The message is a sequence of non-NEWLINE characters (your help message) followed by NEWLINE.

Example:

R	T	F	M	!	\n

Unlike the above example, the HELP message should print out a message which indicates all the commands the user can send to the server. The exact content of this message is up to you – but it is important that this message not contain multiple NEWLINEs.

Problem 1: set up the server to deal with multiple clients

In `minesweeper.server.MinesweeperServer`, we have provided you with a single-threaded 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.

As always, all your code should be safe from bugs, easy to understand, and ready for change. Be sure the code to implement multiple concurrent clients is written clearly and commented if necessary.

You may continue to do nothing with the parsed user input for now.

When you write tests in `MinesweeperServerTest.java`, you may write tests similar to the **published test** we provide, and you may also write tests using `ByteArrayInput/OutputStream` stubs as described in *Sockets & Networking*.

Problem 2: design a data structure for Minesweeper

In `minesweeper.Board`, we have provided a starting file for this problem.

Specify, test, and implement a data structure for representing the Minesweeper board (as a Java type, without using sockets or threads). Define the `Board` ADT as well as any additional classes you might need to accomplish this.

You may ignore thread safety for now.

Your `Board` class (and other classes) must have specifications for all methods; abstraction function, rep invariant, and safety from rep exposure written as comments; and the rep invariant implemented as a `checkRep()` method called by your code.

Problem 3: make the entire system threadsafe

1. Come up with a strategy to make your system thread safe. As we've learned, there are multiple ways to make a system thread safe. For example, this can be done by:

   - using immutable or threadsafe data structures
   - using a queue to send messages that will be processed sequentially by a single thread
   - using synchronized methods or the synchronized keyword at the right places
   - or combination of these techniques

2. The specification for `Board` (and any other classes) should state whether that type is thead-safe.

   In addition to the other internal documentation, `Board` (and any other classes) must document their thread safety argument. Depending on your design, the board (or other classes) may ***not*** be threadsafe. If so, document that.

You will also document your system thread safety argument in `MinesweeperServer.java` in problem 5.

SFB, ETU, RFC: be sure any code to implement thread safety is written clearly and commented if necessary.

Problem 4: initialize the board based on command-line options

We want our server to be able to accept some command-line options. The exact specification is given in the Javadoc for `MinesweeperServer.main()`, which is excerpted below:

> Usage:
>
> ```
> MinesweeperServer [--debug | --no-debug] [--port PORT]
>                   [--size SIZE_X,SIZE_Y | --file FILE]
> ```
>
> The `--debug` argument means the server should run in debug mode. The server should disconnect a client after a BOOM message if and only if the `--debug` flag was NOT given. Using `--no-debug` is the same as using no flag at all.
>
> *E.g.* `MinesweeperServer --debug` starts the server in debug mode.
>
> `PORT` is an optional integer in the range 0 to 65535 inclusive, specifying the port the server should be listening on for incoming connections.
>
> *E.g.* `MinesweeperServer --port 1234` starts the server listening on port 1234.
>
> `SIZE_X` and `SIZE_Y` are optional positive integer arguments specifying that a random board of size SIZE_X × SIZE_Y should be generated.
>
> *E.g.* `MinesweeperServer --size 42,58` starts the server initialized with a random board of size 42 × 58.
>
> `FILE` is an optional argument specifying a file pathname where a board has been stored. If this argument is given, the stored board should be loaded as the starting board.
>
> *E.g.* `MinesweeperServer --file boardfile.txt` starts the server initialized with the board stored in `boardfile.txt`.

We provide you with the code required to parse these command-line arguments in the `main()` method already existing in `MinesweeperServer`. You should not change this method. Instead, you should change `runMinesweeperServer` to handle each of the two different ways a board can be initialized: either by random, or through input from a file. (You'll deal with the debug flag in Problem 5.)

For a **`--size`** argument: if the passed-in size X,Y > 0, the server's `Board` instance should be randomly generated and should have size equal to X by Y. To randomly generate your board, you should assign each square to contain a bomb with probability .25 and otherwise 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 is:

```
FILE ::= BOARD LINE+
BOARD := X SPACE Y NEWLINE
LINE ::= (VAL SPACE)* VAL NEWLINE
VAL ::= 0 | 1
X ::= INT
Y ::= INT
SPACE ::= " "
NEWLINE ::= "\n" | "\r" "\n"?
INT ::= [0-9]+
```

In a properly formatted file matching the FILE grammar, the first line specifies the board size, and it must be followed by exactly Y lines, where each line must contain exactly X values. If the file is properly formatted, the `Board` should be instantiated such that square *i,j* has a bomb if and only if the *i*'th VAL in LINE *j* of the input is 1. If the file is improperly formatted, your program should throw an unchecked exception (either `RuntimeException` or define your own). The following example file describes a board with 4 columns and 3 rows with a single bomb at the location *x=2*, *y=1*.

Example:

4		3	\n				
0		0		0		0	\n
0		0		1		0	\n
0		0		0		0	\n

If neither `--file` nor `--size` is given, generate a random board of size 10 × 10.

Note that `--file` and `--size` may not be specified simultaneously.

Reading from a file: see **tips for reading and writing** below.

Implement the `runMinesweeperServer` method so that it handles the two possible ways to initialize a board (randomly or by loading a file).

Running the server on the command line: this is easiest to do from the `bin` directory where your code is compiled. Open a command prompt, `cd` to your `ps2` directory, then `cd bin`. Here are some examples:

```
cd bin
java minesweeper.server.MinesweeperServer --debug
java minesweeper.server.MinesweeperServer --port 1234
java minesweeper.server.MinesweeperServer --size 123,234
java minesweeper.server.MinesweeperServer --file ../testBoard
java minesweeper.server.MinesweeperServer --debug --port 1234 --size 20,14
```

You can specify command-line arguments in Eclipse by clicking on the drop-down arrow next to "run," clicking on "Run Configurations...", and selecting the "Arguments" tab. Type your arguments into the text box.

Problem 5: putting it all together

1. Modify your server so that it implements our protocols and specifications, using a single instance of `Board`.

   Note that when we send a BOOM message to the user, we should terminate their connection if and only if the debug flag (see Problem 4) is missing.

   The **tips for reading and writing** below may be useful for reading from and writing to socket streams.

2. Near the top of `MinesweeperServer`, include a substantial comment with an argument for why your system is threadsafe.

   SFB, ETU, RFC: be sure the code to implement thread safety is written clearly and commented if necessary.

Tips for reading and writing

- `BufferedReader.readLine()` reads a line of text from an input stream, where line is considered to be terminated by any one of a line feed (`\n`), a carriage return (`\r`), or a carriage return followed immediately by a linefeed. This behavior matches the NEWLINE productions in our grammars.
- You can wrap a `BufferedReader` around both `InputStreamReader` and `FileReader` objects.
- `PrintWriter.println()` prints formatted text to an output stream, and terminates the current line by writing the line separator string. It always writes a line separator, even if the input string already ends with one. The line separator is defined by the system property `line.separator`; you can obtain it using `System.lineSeparator()`. It is `\n` on *nix systems and `\r\n` on Windows. Both of these line endings are allowed by the NEWLINE productions in our grammars.
- You can wrap a `PrintWriter` around a `Writer` or `OutputStream` objects.

Discussion

Topic: Problem Set 2

Show Discussion