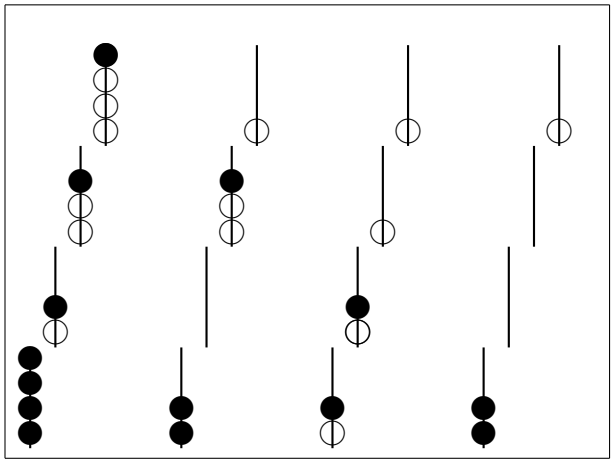

Figure 1: Lines in a Score 4 game.

The board shown to the left has four black lines and two white ones. The position shown could never arise in a real game.

Problem statement

The goal of this project is to write programs to play a 3-dimensional tic-tac-toe game that is sold commercially as "Score 4".[1]

The Game The game is played on a board which consists of a 4×4 grid of pegs (thin metal spikes). On each spike you can slide as many as four coloured beads. There are two players: White and Black. White has 32 white beads, and Black has 32 black beads. The players alternate taking turns, with White playing first. On each player's turn, the player places a bead on one of the 16 pegs that has less than 4 beads already on it.

The first player to get 4 beads in a straight line wins. Should the entire board be filled before either player completes a line, the game is a draw. (In fact, once it becomes clear that it is mathematically impossible for either player to win, the referee can declare the game to be a draw.) A straight line can be horizontal, vertical, or diagonal. Diagonals can be singly or doubly "skewed" as shown in Figure . If you count carefully, you should find that there are 76 lines on a completely-filled board.

Beads cannot hang in the air; they slide down to the bottom of the peg. This makes Score 4 different from $4 \times 4 \times 4$ tic-tac-toe.

Programming tasks

Your team needs to write two or three separate programs: a "referee", a human player interface, and a computer opponent. You may choose to combine the referee and human player interface if you so choose.

The Referee The referee program allows a person to play Score 4 against the computer. It keeps track of whether either player has won the current game; lets the human player quit or restart the game any time (s)he chooses; and draws the board so that the human player can see the current game situation.

ASCII-graphics such as those shown in Figure 2 are acceptable for displaying the board, and were used in the past when this project was done in C++. However, a Graphical User

[1] see `http://www.boardgamegeek.com/game/3656`. It appears that this game was first produced in the 1970's, so patents have likely expired.

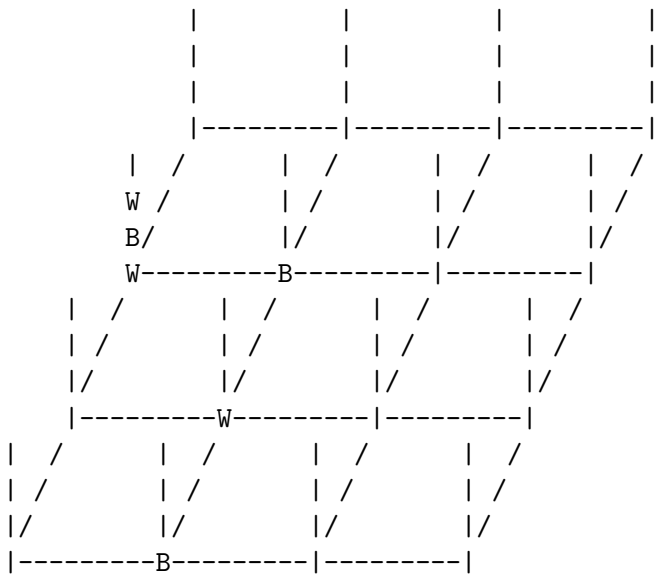

Figure 2: ASCII rendition of Score 4 board.

Interface is preferred. It remains to be seen whether this is feasible as part of a one-semester project.

The exact mechanism that the referee program uses for interacting with the computer opponent will be explained in more detail in a later handout. Generally speaking the referee will send simple commands through an `PrintWriter` to the computer opponent, and the computer opponents response to the commands will appear in an `BufferedReader`.

The Computer Opponent The computer opponent is a stand-alone program that reads commands from standard input (`System.in`) and writes responses to standard output (`System.out`). The computer opponent should make minimal assumptions about the order of the sequence of commands that it receives. The responses should be exactly as described below. Note that the computer opponent must keep track of the current board position.

Commands Each command ends with a semi-colon (**;**). The commands are:

1. "**restart;**". Reset the board to empty and the computer's colour to unknown.
2. "**quit;**". The computer opponent should exit.
3. "**play *colour*;**", where *colour* is either "**white**" or "**black**". From now on the computer should play the colour beads when requested to move.
4. "**move;**". The computer opponent should make a move (and update its copy of the board accordingly). See the responses section below for the format of a move.
5. "**note: *colour* plays *location*;**". Note that the referee has placed a bead of the colour indicated on the board in a position indicated. As above, colour is either "**white**" or "**black**", and the location consists of a letter followed by a number, e.g., "**B1**". The letter is '**A**', '**B**', '**C**' or '**D**'; and the number is '**1**', '**2**', '**3**', or '**4**'.

Responses Each response ends in a period and must be followed by an end-of-line character sequence. The correct multi-platform way to do this is to use a `java.io.PrintWriter`, and then use `.println(...)` or `.format("... %n", ...)`.[2] The response to every command except the move command is "**ok.**"; the response to the move command is a location, e.g., "**A3.**".

General comments

The referee program and the computer opponent program make use of similar concepts, so they should make use of common classes and object files in their construction. *Your coding will be graded in part on how much code is shared between the two program;* as one of the goals of good object oriented programming is to create classes and objects that can be used in multiple program.

For the computer opponent program, correctness is far more important than cleverness. The computer opponent program must work correctly, even when used by a referee program other than your own. However, intelligent play by the computer opponent is not necessary, and should not be a priority when completing the project.

Dr David Casperson
2016-12-06

[2] . . . possibly followed by a `.flush()` The documentation says that `PrintWriter` will take care of the flush when the programmer has indicated a desire to end the line as shown above. Regardless of how it is done, the computer opponent *must* ensure that the response ends in a new line and that its output stream is flushed after sending a response.