Problem Set 7

Due before midnight on Wednesday, November 16, 2016.

1 Assignment Goals

1. Create a multi-class interactive project.
2. Learn how to use separation of responsibility as a design criterion in creating new classes.

2 Problem

The game of Kalaha is described in problem set 5. In that problem set, you implemented three types: `Player`, `BoardFrame`, and `PitFrame`. Together, they represent the structure of a Kalaha board, but they make no mention of the seeds used to play the game.

In problem set 6, you implemented two more classes: `Board` and `Pit`. These classes were responsible for managing the seeds in the pits and for implementing several game-related functions such as `sow()` and `capture()`.

In this problem set, you will complete the Kalaha game by implementing two additional classes, `Game` and `Referee`.

`Game` manages the overall play of the game. It creates and has custody of the `Board` object that is used to play the game. It has a data member to remember who the *active player* is, i.e., the player whose turn it is and who gets to make the next move. It implements several functions that reflect the overall play of the game.

`Referee` is the interface between the user and the abstract game. It creates, manages, and has custody of a `Game` object. It passes messages back and forth between the on-line players and the `Game` object. It displays the board, announces whose turn it is, and prompts the user to enter a move (i.e., a house number). It calls a function `bool Game::isValidMove( int move )` to determine if the user input is a legal move in the current state. A move is valid if the move is a valid house number and that house of the active player is non-empty. If the move is not valid, the referee announces that fact and prompts the user to reenter a valid move.

Once a valid move has been entered, the referee calls a `void Game::doMove(int move)` to perform the move according to the rules of Kalaha. Recall from problem set 5 that a move proceeds in several steps:

1. The starting pit is the one specified by `move`. The seeds are grabbed from the starting pit and sown on the board.

2. At the completion of `sow()`, a capture is performed if applicable.

3. Information recording the result of the move should be stored in additional `Game` data members that the referee can later access with getters. Items of interest to the referee are whether the player gets another turn, whether the move ended in a capture, and if it did, how many seeds were captured. The only use of this information is to inform the user of what happened,

and it need remain valid only until the next call on `Game::doMove()`. The game logic (such as whose turn it is) is maintained in `Game` and is updated before `Game::doMove()` returns. The referee does not try to keep track of whose move it is. Rather, it calls `Player Game::activePlayer()` whenever it needs to know.

When `Game::doMove()` returns, the referee announces what happened and displays the board. It next calls a function `bool Game::isOver()` that determines whether the game is over or not. Recall that the game is over as soon as all of one player's houses become empty.

If the game is not over, the referee loops back to process the next move. If the game is over, the referee calls a function `Game::endGame()` that moves the seeds from the other player's houses to its own store. Finally, the referee announces the number of seeds in each player's store and who won or if it is a tie.

Your solution should make ample use of delegation. In particular, the top-level function `Referee::playGame()` should only talk to public functions in `Game`. You might need to put some functions in `Game` in addition to the ones I mentioned above, for example, to retrieve the game parameters `numHouses` and `numSeeds`, or you might choose to store them in `Referee` as `const int` variables.

A possible sample output is shown in Figure 1. Note that this particular referee was not very talkative and did not announce when a capture took place. Feel free to present the transcript of play in a more pleasing and readable fashion.

The main program should do nothing more than parse the command line arguments (like you did for PS6), instantiate a `Referee`, and pass control to him by calling his `playGame()` function.

3 Testing

One way to test your program is to put a sequence of user inputs into a file, say, `sample.in`, and then run the game with standard input redirected to that file. For example, the sequence of inputs in file `/c/cs427/code/ps7/sample.in` produced the output file `/c/cs427/code/ps7/sample.out` when run with the command line

```
kalaha 6 4 < sample.in > sample.out
```

4 Grading Rubric

Your assignment will be graded according to the scale given in Figure 2.

```
Welcome to Kalaha!

----------
South's turn:
           N1    N2    N3    N4    N5    N6
          [ 4]  [ 4]  [ 4]  [ 4]  [ 4]  [ 4]
  N0 [ 0]                                    [ 0] S0
          [ 4]  [ 4]  [ 4]  [ 4]  [ 4]  [ 4]
           S6    S5    S4    S3    S2    S1

Please enter a pit number for South (q to quit):
South playing move 4
South gets another turn

----------
South's turn:
           N1    N2    N3    N4    N5    N6
          [ 4]  [ 4]  [ 4]  [ 4]  [ 4]  [ 4]
  N0 [ 0]                                    [ 1] S0
          [ 4]  [ 4]  [ 0]  [ 5]  [ 5]  [ 5]
           S6    S5    S4    S3    S2    S1

Please enter a pit number for South (q to quit):
South playing move 6
Turn is over

----------
North's turn:
           N1    N2    N3    N4    N5    N6
          [ 4]  [ 4]  [ 4]  [ 4]  [ 4]  [ 4]
  N0 [ 0]                                    [ 1] S0
          [ 0]  [ 5]  [ 1]  [ 6]  [ 6]  [ 5]
           S6    S5    S4    S3    S2    S1

Please enter a pit number for North (q to quit):
North playing move 4
North gets another turn

----------
North's turn:
           N1    N2    N3    N4    N5    N6
          [ 5]  [ 5]  [ 5]  [ 0]  [ 4]  [ 4]
  N0 [ 1]                                    [ 1] S0
          [ 0]  [ 5]  [ 1]  [ 6]  [ 6]  [ 5]
           S6    S5    S4    S3    S2    S1

Please enter a pit number for North (q to quit):
North playing move 6
Turn is over
```

Figure 1: Initial part of a game transcript.

#	Pts.	Item
1.	1	A well-formed `Makefile` or `makefile` is submitted that specifies compiler options `-O1 -g -Wall -std=c++11`.
2.	1	Running `make` successfully compiles and links the project and results in an executable file `kalaha`.
3.	4	`kalaha 6 4` when run on the sample input file results in the same final board as shown in the sample output file.
4.	4	`kalaha 3 2` gives correct answers when tested on an input file `mytest_3-2.in` of your choosing that leads to a complete game.
5.	4	Classes from PS6 are used by the new classes but not changed. Any exceptions are noted and explained in a submitted notes file.
6.	4	All game logic resides in `Game` (and `Board`). All user interaction interaction is initiated and controlled by `Referee`.
7.	1	All of the instructions in section 2 are followed.
8.	1	All relevant standards from previous problem sets are followed regarding good coding style, submission, identification of authorship on all files, and so forth.
	20	Total points.

Figure 2: Grading rubric.