CPSC 231: Computer Science II
Fall 2020
Mastery Project 3: Modern War(fare)
Due: November 17, 2020 at 11:59 pm

Reading

This assignment assumes you have completed the reading for weeks 1-10 on zyBooks.

Being Upfront...Up Front

We are well aware that this assignment and its solution has been disseminated widely on the internet on sites such as Chegg.com. Last semester we had to penalize some individuals for copying code from these solutions. In cases where these solutions were not given attribution, this involved formal reports to the academic integrity committee.

For this assignment, you are prohibited from searching for, looking at, or sharing any code available publicly or privately that is related to the solution for this assignment. We will be using MOSS (http://theory.stanford.edu/~aiken/moss/) to check all code submitted for this assignment against solutions available online as well as submitted during the previous 3 semesters. Any evidence of copied code from these sources will result in an academic integrity violation with a grade of "F" in the course. Because the instructions for this assignment explicitly prevent referring to such sources IN ANY CAPACITY, we will take action even if this code is given proper attribution. Thus, if you reference these sources for this assignment, you are better off not submitting your work and receiving a 0 than you are submitting work that resembles previous solutions.

Note that this assignment has also been changed from previous semesters, so it will be immediately obvious if previous solutions are copied.

The Assignment

Now that you know how to write OO Java code let's put your skills to use in your first large (multiple class) programming assignment. Your task for the next week is to write a simulation of a modified version of the card game War. We call it...Modern War(fare).

In this game of War, two players battle each other to see who can win the most cards. At the start of a single game a deck of 52 cards (legal values 2-10, J, Q, K, A) is split evenly between the two players such that each player holds a stack of 26 random cards. For each battle of the game both players turn over three cards at the top of their stack, and compute the median (middle) value of the cards. If the player has fewer than 3 cards, they take the maximum value. The player with the higher value based on the median (or max value for a player with less than 3 cards) takes all cards played in the battle and places them on the bottom of their stack. If the two values played are equal (a "war"), each player lays down one card and the higher-valued card wins all of the cards on the table, which are then added to the bottom of the player's stack. In the case of another tie, the war process is repeated until there is no tie.

A player wins by collecting all the cards. Once a single player holds all 52 cards, the game ends.

You are to write a program that takes the number of games, *n,* to play as a command line parameter (get passed in the String[] args parameter to main, not from standard input) and then simulates the *n* games. At the conclusion of all the games your program will calculate and output the following statistics:

Average number of battles per game
Average number of wars per game
Average number of double wars per game
Max number of battles in a game
Min number of battles in a game
Max number of wars in a game
Min number of wars in a game

Designing a Solution

As you can probably tell, this assignment is fairly involved. In fact, the only way you can pull it off is if you are smart about the abstractions you use and how you break your program into separate classes. It is recommended that you design the following classes in implementing your solution:

Card

A card consists of a value (low to highest: 2-10, J, Q, K, A) and a suit (hearts, spaces, clubs, diamonds). Provide appropriate constructors as well as accessor methods.

Deck

A new deck consists of all 52 cards in a LinkedList. It contains a method, *deal*, that removes a random card from the list and returns that card. (This can be achieved by generating a random number between 0 (inclusive) and the length of the list (exclusive) and removing/returning the card at that list position.

Player

A player consists of a number (1 or 2) and a LinkedList of owned cards. At the start of the game each player holds 26 unique cards. A method, *flip,* removes and returns the cards at the front of the list. A method*, collect*, takes a collection of cards as a parameter and adds it to the list of owned cards. The player class should contain the appropriate constructors and other helper methods as required. For example, a *hasWon* method would return true if a player owned all 52 cards, and false otherwise, and a method, *war*, that would handle what the player does when a war happens.

Game

A game consists of two players, as well as other instance variables for keeping track of the statistics for each game. A method, *play*, carries out the rules of the game of war until one of the players has won, and a method, *war*, might handle the logic for breaking ties. This class should have a default constructor and accessor methods, as well as well as any other methods you feel are appropriate.

Simulation (this is your Driver class)

This class contains a constructor that takes the number of games to simulate as well as a function, *simulate*, that plays the specified number of games. A method*, calculate,* computes the aggregate statistics from all games, which will require you to define the appropriate class member variables to keep track of the results of each game. Another method*, report,* prints the required statistics to the screen in a nicely-formatted manner. The main method for running your War program is contained in this class.

Note that it is possible to get into an infinite loop if the cards happen to always be collected in the same order. As an easy fix, when a player collects cards and adds to the bottom of the stack you can add them in random order.

WarLogger
The WarLogger class is provided for you and is used to log the internal details of your game for grading purposes, but other than calling its methods you need not be concerned with it. **DO NOT EDIT THIS CODE FOR ANY REASON.** Generate the Javadoc for the class to see details for the methods. You should use the WarLogger for the following:

After a Battle is Dealt
After a battle is dealt (cards given to both players), you should call for each player: WarLogger.getInstance().logBattle(…) passing in the correct parameters. Note there are constants defined in the WarLogger class to represent player 1 and player 2. For example, to log the hand dealt for battle number 10 for player 2, you would call: WarLogger.getInstance().logBattle(10,WarLogger.P2,h) where h is an array* of cards held by player 2 for the battle.

*note: check the LinkedList API on how to handle an Array here

After a Battle is Won
To record the outcome of a battle call WarLogger.getInstance().logBattleOutcome(…) passing in the correct parameters.

After a War is Won
To record the outcome of a war call WarLogger.getInstance().logWarOutcome(…) passing in the correct parameters.

After a Game is Won
To record the outcome of a game call WarLogger.getInstance().logGameOutcome(…) passing in the correct parameters.

At the End of Your Simulation
THE LAST LINE OF YOUR MAIN METHOD IN THE SIMULATION CLASS SHOULD BE:
WarLogger.getInstance().release();

Being OO
It is expected that your class implementations adhere to OO best practices. Methods should be given the correct access specification (public or private) depending on their purpose and use. Appropriate constructors should be provided for each class. Accessor/Mutator methods should be provided when there is a need to return/alter member variables.

You are of course free to add as many classes/methods as you like to get the job done, but keep in mind that you should strive for an elegant, efficient solution.

Due Date

This assignment is due at 11:59 pm on 11-17-2020. Submit the appropriate zip file to Canvas, it should be named LastNameFirstInitial_MP3. Please make sure to include all the required files (README, source files).

Grading

Assignments will be graded on correctness, adherence to style, and the inclusion of meaningful comments. Refer to this URL for a widely used Java style guide: https://google.github.io/styleguide/javaguide.html