CSc 345 — Analysis of Discrete Structures
Spring 2019 Section 2 (Anson)

Program #2: Bones Battle

Due Date: February 28th, 2019, at the beginning of class

Overview: In 2006 a Flash game called Dice Wars appeared. When it caught my attention, I had two thoughts: (1) 'This could make a good 345 assignment,' and (2) 'This game cheats!' I wrote a version in Java that doesn't cheat. In this assignment, you'll be writing most of the 'behind the scenes' code for our version, called "Bones Battle."

You can think of Bones Battle as a simplified version of the board game Risk. Our version has two to five players, distinguished by name and color. All but one of the players is played by the computer, with the human controlling the fifth (green) player. The game board is a connected arrangement of colored squares (territories), each labeled with an initial quantity of 6-sided dice. A turn consists of 0 or more attacks, each from a square with 2 or more dice to an adjacent (sharing a side) opponent's square. If the roll of the attacker's dice total more than the opponent's total, the attacked square's ownership transfers to the attacker and all but one of the attacking dice are moved to it. Otherwise, the attacking square loses all but one of its dice. These totals are briefly displayed on the attacking/defending squares. Totals in white mean that the attacker won, while magenta means the defender prevailed. When a turn is complete, the computer determines the quantity of squares in the largest connected cluster of squares owned by the player. A number of dice equal to this quantity is added randomly to the player's squares, to a maximum of eight per square.

When a computer-controlled player is playing, the human player merely observes the action described above. When it's the human player's turn, s/he begins an attack by clicking a legal attacking square (its color darkens to reflect the selection). Next, s/he clicks the adjacent opponent square to be attacked. From there, the computer will complete the attack. When the player is finished selecting attacks, s/he clicks 'Next' and the next computer player takes its turn.

My version of Bones Battle is available as a JAR (Java ARchive) file, if you'd care to play it. See the 'Want to Learn More?' section, below, for details.

Assignment: In this assignment you'll be writing a significant portion of the Bones Battle code, specifically the four Java classes Territory, Map, Graph, and ComputerStrategy. Territory represents a square on the game board, Map handles interactions with the game board, using the Graph class to represent the collection(s) of territories and their relationships. ComputerStrategy handles the logic of the computer-controlled players.

Please note that the following describes what you **must** include in these classes. If you wish to add more (e.g., additional private methods), you may, so long as the additions fit within the existing framework.

1. *The Territory Class:* Each of the squares on the game board is represented within the game by a Territory object. A Territory needs to know the map with which it is associated (type: Map), the quantity of dice currently assigned to it (int), its identification number (int; see below), and a reference to the player who owns it (Player).

   - Constructor #1: *Territory (Map map)*

     Sets the map reference to the argument, the owner reference to null, and the integers to -1.
   - Constructor #2: *Territory (Map map, Player owner, int dice, int idNum)*

     Sets the state variables to the values of the given arguments.
   - The Accessors ("Getters"):

     int getDice (), *int getIdNum ()*, *Map getMap ()*, and *Player getOwner ()*.
   - The Mutators ("Setters"):

void setDice (int), *void setIdNum (int)*, and *void setOwner (Player)*.

- Other Methods:
  - *int getRow ()* and *int getCol ()*: A territory's row and column coordinates within the game board can be computed from the territory's ID# because the ID#s must be assigned as follows: The territory assigned to the upper left corner (row 0, column 0) has ID# 0. The territory to the right of ID# 0 has ID# 1, and the territory just below ID# 0 is ID# 8, assuming that the board has eight columns and at least two rows. In other words, the territories are numbered in row-major order. Knowing that, the row and column indices can be easily computed from the ID#.

2. *The Map Class:* This provides the representation of the game board – a rectangle of territories – with several 'holes' (invisible, unplayable territories) to make the board more interesting.

   A Map state consists of the map (`Territory[][]`), a reference to a graph data structure that knows neighboring territories (see the Graph class, below), and a list of references to the game's players (`ArrayList<Player>`).

   - Constants (all of type int):
     - *ROWS* and *COLUMNS*: Size of the board.
     - *VICTIMS*: Number of unused ('invisible') territories.
     - *NUMTERRITORIES*: Synonym for ROWS times COLUMNS.
     - *OCCUPIED*: Synonym for NUMTERRITORIES minus VICTIMS.
     - *MAXDICE*: The largest quantity of dice a territory may have.
   - The Constructor: `Map (ArrayList<Player> players, int rows, int columns, int victims, int maxDice)`

     The arguments are used to set the corresponding constants and instance variables. The constructor also declares the map array (that is, the game board), and creates territories for the board (w/ correct ID#s). It also creates the territory neighbor graph by calling the `constructGraph()` method and concludes by calling the local `partitionTerritories()` and `distributeDice()` methods (which you'll write; see below) to initialize the game board.
   - The Accessors ("Getters"):

     Territory[][] getMap (), *Graph getGraph ()*, and *Territory getTerritory (int row, int column)*.

     getTerritory simply returns the Territory reference stored in the game board at location (row, column).
   - The Mutators ("Setters"): There are no mutators in this class.
   - Additional Methods:
     - *public int getTerritoryId (int row, int column)*: Given the row and column of a territory, compute and return the territory's ID#. The Territory class supplies getRow() and getCol() methods; see "The Supplied Classes" section, below.
     - *public int countTerritories (Player player)*: Determine and return the quantity of territories owned by the given player.
     - *public int countDice (Player player)*: Determine and return the total number of dice currently assigned to this player's territories.
     - *public ArrayList<Territory> getPropertyOf(Player player)*: Construct and return a reference to an ArrayList of Territory object references. The territories referenced by the list are those currently owned by the given player.
     - *public ArrayList<Territory> getNeighbors(Territory cell)*: Each territory has at least one adjacent (edge-sharing) neighboring territory, but no more than four. This method returns a reference to an ArrayList of references to the given territory's neighbors. The Graph object offers a helpful method: isInGraph(). isInGraph() takes a territory ID# and returns true if the territory is participating in the game (remember, some territories are 'invisible' and thus can't be neighbors).
     - *public ArrayList<Territory> getEnemyNeighbors(Territory cell)*: Similar to getNeighbors(), above, but the returned list contains only references to neighboring territories controlled by another player.

    - *private void partitionTerritories()*: This method is called by the constructor after the array of un-owned territories has been built. This method assigns to each player the same (or nearly the same) quantity of territories. Each player's territories are to be selected randomly. That is, do not assign players to territories based on a pattern.
    - *private void distributeDice()*: This method is called by the constructor after partitionTerritories() has been called. Collectively, the assigned territories of each player have the same quantity of dice as the territories of every other player: three times the player's number of territories. For example, if there were three players and 15 territories per player, each player would start with 45 dice. This method randomly distributes each player's dice allotment across his or her territories. Each territory must have at least one die, but no territory can have more than MAXDICE dice.
    - *public int countConnected (Player player)*: Returns a count of the number of territories in the largest connected cluster of territories owned by the given player.
    - *public Graph constructGraph(int rows, int cols, int victims)*: Builds and returns a reference to a graph representing all of the active territories in the game. An acceptable graph has the appropriate number of active territories (Map.OCCUPIED), 'scatters' the inactive territories among the active territories in an unpredictable (pseudo-random) fashion, and ensures that all of the active vertices are connected.

3. *The Graph Class:* The Map class relies on a Graph object to represent the neighbor relationships between territories. Each territory is represented by a corresponding vertex within the graph, and neighboring territories are connected by edges. A Graph object needs a representation for the graph, and a way to know which of the vertices are considered to be 'inactive' by the game.

   The graph vertices need to be numbered in the same way that the game numbers the territories: The territory assigned to the upper left corner (row 0, column 0) of the board has ID# 0. The territory to its immediate right has ID# 1, and the territory just below ID# 0 has ID# 8. In other words, the territories, and thus the vertices, are numbered in row-major order. Knowing that, the row and column indices can be easily computed from the ID#, and vice-versa.

   - The Constructor: *Graph (int numVertices)*

     The number of vertices in the graph is supplied. The constructor will create a suitable graph representation that initially includes no edges. Because our application requires that some of the vertices be 'inactive' within the game, the graph needs to be able to distinguish the inactive vertices from the active vertices.
   - Additional Methods:
     - *public List<Integer> getUnusedVertices ()*: Returns a list of the ID#s of the inactive vertices of the graph. If there are no inactive vertices, a reference to an empty list object is returned.
     - *public boolean isEdge (int source, int destination)*: Returns true if the graph possesses an edge directly connecting the given source and destination vertices. That is, 'true' is returned if these vertices are adjacent.
     - *public void addEdge (int source, int destination)*: Ensures that the graph contains an edge connecting the given source and destination vertices.
     - *public void removeEdge (int source, int destination)*: Ensures that the graph does not contain an edge connecting the given source and destination vertices.
     - *public boolean isInGraph (int vertex)*: Returns true if the given vertex is active within the graph.
     - *public void removeVertex (int vertex)*: Called to mark a vertex as inactive. Inactive vertices have no neighbors, and thus have a degree of 0.
     - *public List<Integer> getAdjacent (int vertex)*: Returns a list of vertex ID#s. A vertex is in the list if it is active and adjacent to the given vertex.
     - *public int degree (int vertex)*: Returns the degree of the given vertex.
     - *public boolean connected ()*: Returns true if the graph is connected; that is, if every active vertex is reachable from every other active vertex.

4. *The ComputerStrategy Class:* When you study the demo version of the game, you'll find that the computer players make attacks that, rather than being evocative of Sun Tzu's classic "The Art of War,"

remind you more of "Zapp Brannigan's Big Book of War."[1] Here's your chance to write a strategy that will make beating the computer a real challenge for the human player. The methods that you need to implement are listed in the Strategy.java interface, an interface that your ComputerStrategy class must implement.

The strategy class used in the demonstration game is not very strategic at all. All it does is ask, "Is there an attacker-defender pair in which the attacker's dice quantity is greater than or equal to that of the defender?" If such a pair can be found, it recommends attacking. All we require of your strategy is that it be more intelligent than that. For example, here's a somewhat brighter strategy: Give preference to attacks that, if successful, would connect two disconnected clusters of the player's territories. There are many more options, and of course you are encouraged to create a very complex strategy if you wish to do so.[2]

- The Constructor: Don't write one! (Why not? For dynamic class loading to work in tournament mode,your strategy class can have only the default constructor.)
- The Mutator ("Setter"):
  - *public void setPlayer (Player whom)*:
    The strategy object needs to know on which player's behalf it is thinking strategically. If we could have a constructor, we wouldn't need this setter.
- Additional Methods:
  - *public boolean willAttack (Map board)*: Returns true if the player will attack, given the state of the game as described by the current game board. As this determination is made, the method may construct data structures whose content can be used to assist in the processing of getAttacker() and getDefender().
  - *public Territory getAttacker ()*: Returns a reference to a territory that will be the source of the next attack. The selected territory must have at least two dice and be owned by the player associated with this strategy object. This method should be called only after willAttack() has been called.
  - *public Territory getDefender ()*: Returns a reference to the territory object that will defend the attack of the territory just identified by getAttacker(). The defending territory must be occupied (owned) by another player and must be adjacent to the attacking territory. This method should be called only after getAttacker() has been called.

The collection of files for this assignment includes a very trivial strategy class, `MilquetoastStrategy`, as an example.

5. *The "Your NetID Here" Class:* As added incentive to write a superior computer strategy, some time after the due date we'll have an in-class contest that will pit your strategy against those of the other students. Built-in to the Bones Battle game is a 'tournament mode' in which strategies can play one another until a given number of victories by one strategy is achieved.

   To run Bones Battle in this mode, add the `.class` file names of the strategies to be used in the game to the command line. For example:

   ```
   java Bones Zapp Fry Leela Zoidberg
   ```

   Up to five strategy classes can be listed. By default, in tournament mode the game will run at an accelerated speed, will report victories to the terminal window, and will stop when a strategy has accumulated 16 victories. The winning total can be adjusted with the command line argument `wins=W`; just replace the `W` with the desired win total. Use `speed=M` to change the pace of the game, where `M` is the number of milliseconds per attack. Unless changed, `M` = 20ms in tournament mode, a hundredth of the delay of normal play.

   Constructing a strategy for tournament play is easy; just make a copy of your ComputerStrategy.java file named with your capitalized lectura login name. For example, if your login name is 'zapp,' copy the strategy file to 'Zapp.java' (and in the copy change the name of the class to 'Zapp' as well, of course).

[1]Futurama, "Love's Labors Lost in Space," Season 1, Episode 4
[2]If your strategy ends up enslaving humanity, we reserve the right to retroactively fail you. ☺

Your strategy code should be entirely contained within the file; that is, no auxilliary .java files should be used.

Please note that success in the tournament will have no bearing on your score for this assignment, or for your grade in this class, for that matter. However, you should try your strategy in the contest environment, as this is a nice way to further test the correct coding of your strategy class. And, submitting a tournament-ready strategy class is a requirement for this assignment, so please don't forget to create it and submit it.

The Supplied Classes: The classes you will be writing are just a portion of the game. Because the knowledge can help, here's a summary of the pre-compiled classes we are supplying:

- *The Bones Class:* This is the game engine. As it calls methods of other classes, rather than other classes calling its methods, its workings aren't important for this assignment.
- *The Player Class:* Each player in the game is represented as a Player object. A Player object knows the player's internal ID number (of type int), name (String), color (Color) (and 'clicked' color (Color)), and has a reference to the strategy this player uses to select attacks (Strategy).

  The constructor and public methods of Player are: *Player (String name, Color color)*, *int getId ()*, *String getName ()*, *Strategy getStrategy ()*, *Color getColor ()*, *Color getClickColor ()*, *void setName (String)*, *void setStrategy (Strategy)*, *void setColor (Color)*, *void setClickColor (Color)*, *boolean willAttack (Map)*, *Territory getAttacker ()*, and *Territory getDefender ()*. (The last three exist to allow the game to talk to the player's associated strategy object.)
- *The StrategyLoader Class:* Like the `Bones` class, this one you don't need to worry about. StrategyLoader is what makes dynamic loading of strategy classes, and thus the tournament mode, possible.

Input: None of the classes you are to write accept input from the outside world; they each exist only to support the Bones Battle game. Thus, we cannot supply any input data for testing.

We have, however, supplied the rest of the `.class` files that comprise the Bones Battle game. A ZIP file containing them (`prog2.zip`) is linked to the class web page. It also includes the `Strategy.java` interface file, the sample `MilquetoastStrategy` source code, as well as a `ReadMe.txt` file that explains how to use the supplied classes with your compiled classes to assemble the complete game. Finally, as mentioned above, an executable version of the complete game is available; more information on it is provided below.

We will test your classes by (a) plugging them into the normal game to verify that the game works, and (b) writing programs that exercise the methods to verify their correct operation. We strongly recommend that you do both of these, too. As always, you may **NOT** share your assignment code with other students, but you may share your testing programs. Just keep in mind that testing programs can have bugs, too. And, remember that you can test your strategy by pitting it against those of your classmates.

Output: As these classes produce no output of their own, we have no output specifications to give you. See the Input section, above, for details on how we will test your classes.

Hand In: On the due date, submit your completed `.java` program files (and your tournament-ready strategy's `.class` file) using the `turnin` facility on lectura. Do not zip them up! The submission folder is `cs345s19_p2`.

Want to Learn More?

- Want to play the original Dice Wars? Visit: `http://www.gamedesign.jp/flash/dice/dice.html`.
- Want to play Bones Battle before you code it? Download the `BonesBattle.jar` file from the class web page, and run it from the command line with `java -jar BonesBattle.jar`. Note: Tournament mode isn't available in this JAR-red version of the game.
- Why the name 'Bones Battle?' Because early dice were sometimes carved from bone, dice are sometimes referred to as 'bones' (as in the phrase "roll them bones!").

Other Requirements and Hints:

- $\boxed{\textbf{Start early!}}$ You will be writing a fair amount of code for this assignment, and the classes you write will not only have to integrate with each other, they'll have to work with the supplied classes to complete the game. This usually means there will be lots of mystery Java exceptions to figure out. 2 a.m. on the due date is a lousy time to be stuck doing that.
- Your first order of business: Play the game! It's hard to code what you don't fully grasp.
- You don't have to (nor should you!) rely only on running the game to test your classes. **We encourage you (strongly!) to write your own testing code.** Note that you may share testing code with other students, but don't share the assignment code.
- The game can run your strategy against itself in tournament mode; it's not as much fun as running it against other strategies, but it does work, and it's easy. Just list your strategy class name multiple times on the command line.
- A reminder about program documentation: *Document your code as you write it.* If you wait until the code's all working before you document it, you'll be spending hours doing nothing but writing documentation. Most people find hours spent in this way to be a trifle boring. Instead, when you start writing a class, write the class block comment first. As you write each method, document it. As soon as you declare a variable, give it a good name and a brief comment. This comment-as-you-code approach is not only less boring, you're also more likely to remember decisions that went into your coding (e.g., choice of algorithms) that should be mentioned in the documentation. Remember, the point is to make happier the poor schmuck who will someday have to read your code in order to understand and modify it. Keep in mind that the poor schmuck may well be the future you.

  Speaking of documentation: Be sure to give a detailed description of the strategy that your ComputerStrategy class implements. The appropriate place for this is in the class block comment at the top of your ComputerStrategy.java file.
- When running java on lectura, be sure that you're using Oracle's Java. When you ask java for its version (`javac -version` and `java -version`) on the department machines, it should report the version to be `javac 1.8.0_101`. If you see a different version, you'll want to give the complete pathname to the compiler and interpreter: `/usr/bin/javac` and `/usr/bin/java`, respectively. Mixing .class files created by different versions of Java usually produces bizarre compilation errors.
- Using a Mac and getting no colors when running locally? Add this flag when you run the program:
  `-Dswing.defaultlaf=javax.swing.plaf.metal.MetalLookAndFeel`