Project 4

CPSC 2150
Part 1 Due: Sunday, November 13th at 11:59 pm
Full Project Due: Sunday, November 20th at 11:59 pm
100 pts

Introduction:

We will continue to work on our Extended Tic-Tac-Toe game for this assignment, but we will not be adding any new functionality to our program. We will create automated test cases for our `IGameBoard` interface and implementations using JUnit. You do not need to create test cases for the `GameScreen` class.

Requirements:

You will create two JUnit test classes, one for each implementation of the `IGameBoard` interface. These classes should be named `TestGameBoard` and `TestGameBoardMem`. The test cases can be exactly the same in each class; you just need to change the implementation used. If you create a `private` helper method to call the constructor as we did in the lab (Factory Method Pattern), this will be an easy change.

You must test the following methods in each JUnit test class. Each test case should have its own JUnit test function.

Test the following methods:

Constructor
- Create 3 distinct test cases for the constructor

`checkSpace`
- Create 3 distinct test cases for `checkSpace`

`checkHorizontalWin`
- Create 4 distinct test cases

`checkVerticalWin`
- Create 4 distinct test cases

`checkDiagonalWin`
- Create 7 distinct test cases
- **Note:** the different diagonals are distinct

`checkForDraw`
- Create 4 distinct test cases

`whatsAtPos`
- Create 5 distinct test cases

`isPlayerAtPos`
- Create 5 distinct test cases

`placeMarker`
- Create 5 distinct test cases

Your JUnit test code must also use the Factory Method Design Pattern by having a `private` method that calls the constructor for your `IGameBoard` object and returns it. By always using the

Factory Method, you will complete the JUnit code for `TestGameBoard`, then copy and paste the code and change only this helper method and the file name to create `TestGameBoardMem`.

You will also need an additional `private` helper method that will take in a 2D array of characters and return a string representation of that array. This string representation should exactly match what the `toString` method returns from our `GameBoard` classes. This will allow you to create an "expected" version of the `GameBoard` in a 2D array, then get the string version to compare to the actual output of your `GameBoard`. So your process for comparing your expected `GameBoard` and your actual `GameBoard` will look like this:

1. Create an empty "expected" 2D array
2. Call the factory method to create your empty `IGameBoard` object "`gb`"
3. For each token, you need to place for your test case
    a. Use `placeMarker` to place it on your `IGameBoard` object "`gb`"
    b. Place that same character in the same location in your 2D array "expected"
4. Call your helper method to create the expected string version of your 2D array
5. Call `assertEquals` to ensure your expected string and `gb.toString()` are equivalent

The Program Report

Along with your code, you must submit a well-formatted report with the following parts:

Requirements Analysis

Fully analyze the requirements of this program. Express all functional requirements as user stories. Remember to list all non-functional requirements of the program as well. Creating test cases is not a functional or non-functional requirement, so these should be the same as previous assignments.

Design

Create a UML class diagram for each class and interface in the program. Also, create UML Activity diagrams for every method in your `GameScreen` class and every method in your `GameBoard` or `GameBoardMem` class. Suppose a method is defined as a `default` method in the interface. In that case, you need to update the diagrams to remove any mention of the 2D array or other private data of the `GameBoard` class and instead refer to primary methods. If a method is provided only as a `default` method in the interface, you only need to provide an activity diagram for the `default` method; you do not need to include the diagram for each implementation. If you provide a `default` method in the interface but override it in the implementation, you must include an activity diagram for each method. Your diagram for `toString` should not reference private data since it will exist in the abstract class.

As software engineers, we use UML diagrams not just as a design tool but as a way to document our code, so other software engineers have reference materials to help them when they need to maintain, fix, test, or build off of our code.

You do not need to create diagrams for your test classes that you will make for this assignment, so these diagrams are the same as the previous assignment.

Testing

In your report, include the test case method name and description for each test case which includes your input values, your expected output, and a reason for why you chose this test case and what

makes it distinct. Remember that the current state of the `GameBoard` is part of the input and part of the output. Follow the same format as outlined in the example test case PDF.

Deployment

Provide instructions about how your program can be compiled and run. Since you are required to submit a `makefile` with your code, this should be pretty simple. Your `makefile` should include targets to compile your program (`default`), run your program (`run`), compile your test cases (`test`), and run your test cases (`make testGB` and `make testGBmem`). Your `makefile` must work on the School of Computing Unix machines. Your `classpath` for JUnit may be different on your personal machine. Make sure you are using JUnit 4. Your `makefile` is more important when it comes to testing (hard to run test cases won't ever be used), so it is worth more points for this assignment.

Additional Requirements and Tips

- You must include a `makefile` with your program. We should be able to run and test your program by unzipping your directory and using the `make`, `make run`, `make test`, `make testGB` and `make testGBmem` commands.
- Your `makefile` must work on the School of Computing Unix machines. Your `classpath` for JUnit may be different on your personal machine.
- Make sure you use JUnit 4 on the School of Computing Unix machines.
- If you discover any failures with your test cases, you will need to find the underlying fault and correct it.
- Your UML Diagrams must be made electronically. I recommend the freely available program Diagrams.net (formally Draw.io), but if you have another diagramming tool you prefer, feel free to use that. It may be faster to draw rough drafts by hand, and then when you have the logic worked out, create the final version electronically.
- **Remember:** JUnit code isn't production code, so hardcoded values are acceptable (and necessary) for our input and expected output values in our JUnit code.
- You will need to call several methods in order to set up your test case. That is fine, as long as your `assertEquals` (or `assertTrue`) statements only test one method. Remember, if you run your test cases and have failed test cases for `whatsAtPos` and `checkDiagonalWin`, you want to fix `whatsAtPos` first, as that method is called in `checkDiagonalWin`.
- You do not need to follow the entire process in your `main` function to test your `IGameBoard` class because our input is hardcoded and always follows our preconditions. This means
    - You aren't asking the user for input
    - You aren't validating input
    - You don't need to call `checkSpace` before placing tokens since they are hardcoded, and we know the position is free
    - You don't need to call `checkForWinner` after every token placement
    - You aren't printing anything to the screen
    - You don't need to follow the turn order when placing tokens.
- **Remember:** Test cases should be distinct in a meaningful way. The difference between checking for a win with the exact same board but swapping `X's` and `O's` is not distinct, as `IGameBoard` doesn't actually care what the character is. Two test cases are distinct if there is a reasonable expectation as to why one test case could pass while the other could fail.

- You do not need to use any specific method (such as path testing) to identify test cases.
- Remember to think about what was challenging for you while writing code. What mistakes did you make in earlier attempts? Those would be good scenarios to test for.
- The JUnit code for the test cases will be very similar for each method. Ensure you get one example working, then copy and paste to change the input and expected output values.
- Remember to follow the best practices we discussed for JUnit test cases. They are different from the best practices for our standard production code.
- You may need to use several `assertEquals` (or `assertTrue`) statements to check if everything worked correctly by checking the string representation, the return value of the method, the getter methods, and so on.
- **Make sure that your JUnit test cases match what you have described in the project report. The TAs will use the project report to check that you have correctly implemented it with the defined input and expected output values.**
- While writing JUnit code is not necessarily difficult, it is time-consuming. Thinking of all the test cases and fixing any faults you find will also take some time. Avoid the temptation to put this assignment of because it's "just" writing test cases.
- **You should not create test cases that violate the preconditions for the method being tested.**

Before Submitting

You need to make sure your code will run on SoC Unix machines and create a `makefile`. Include your name as a comment at the top of each Java file.

Use the command "`zip -r Project4.zip cpsc2150 doc makefile report.pdf`" to zip up all the code, `makefile`, and project report. Don't forget the `-r` to make sure it includes the subdirectories as well. **NOTE:** Copying and pasting this command may cause issues because Windows encodes the − symbol differently from Unix.

Submission:

To ensure that you do not wait until the last minute to start the assignment, we are requesting two submissions for this assignment. There will be separate links to submit your files on Gradescope by the specified date. Even a minute after that deadline will be considered late. Make sure you are prepared to submit earlier in the day, so you are prepared to handle any last-second issues with submission. **Late submissions will not be accepted.**

For part 1, you will update your project report PDF and submit your test plan for all the test cases you plan to implement to Gradescope. This part will be due Sunday, November 13th at 11:59 pm.

For the full project, you will submit your source code, JUnit files, Javadoc files, `makefile`, and project report PDF. The full project is due Sunday, November 20th at 11:59 pm.

Upload your `Project4.zip` to Gradescope. It should automatically check if you have submitted everything correctly and verify that it was able to compile and test your code. While not all test cases are visible, you should make sure that all visible test cases pass. The visible test cases should be a good starting point for checking whether your solution is correct or not.

This is an INDIVIDUAL assignment. You should not be working on this assignment with any other student.

NOTE: Always check your submissions on Gradescope to ensure you uploaded the correct files and it is in the proper format that it expects. **I wouldn't wait until the last minute to submit as Gradescope could take a few minutes to display the results.** Notify your course instructor immediately of any Gradescope issues.

Disclaimer:

It is possible that these instructions may need to be updated. As students ask questions, I may realize that something is not as straightforward as I thought, and I may add detail to clarify the issue. Changes to the instructions will not change the assignment drastically and will not require reworking code. They would just be made to clarify the expectations and requirements. If any changes are made, they will be announced on Canvas.