Project 5

CPSC 2150

Due: **Monday**, April 25th at 11:59 pm

100 pts

Introduction:

In this assignment we will be using a Graphical User Interface (GUI) and the Model-View-Controller architectural pattern to move our Extended Connect X game from a command line interface to a Graphical User Interface. Luckily, much of the work is already done for us, as Project 3 provided the **Model**, and I am providing the **View**. You will just need to complete the **Controller** for our program to work. **You will also be generating a complete Javadoc for your assignment.**

Instructions:

1. Download the `ConnectXGui.zip` file from Canvas.
2. Extract (decompress) the file on your computer.
3. This file contains the entire project directory for an IntelliJ project.
4. Open IntelliJ and use file->open to select your decompressed directory. The project will load in IntelliJ.
5. Take some time to familiarize yourself with the provided code. You will not need to change most of these files, but you will still need to understand how they work and what public methods they offer in order to complete this assignment. See the file descriptions below
6. Add your `BoardPosition` class to this package (make sure the package matches)
7. Add in your `IGameBoard` interface, the `AbsGameBoard` abstract class, and both implementations to this package.
8. As long as your `BoardPosition`, `IGameBoard` and implementations were correct from project 3 and 4, you should not need to make any changes to these files. These files will serve as the **Model** in our Model-View-Controller architectural pattern. One important note, there should be absolutely no statements to print to the screen or get input from the command line interface, as that interface is no longer being used in our program.
9. You do not need to add in your `GameScreen` code, or the test files to this project.
10. You are now ready to begin completing the assignment. See the instructions below on what files you should edit.
11. Make sure to thoroughly test your game, add comments.
12. When you are done, zip up the entire directory and submit that. We will not be able to grade this assignment on SoC Unix or Gradescope, so we will need the entire project directory.

Contracts, Comments, and Formatting

In general, your code should be well commented. Your Javadoc comments will handle a lot of this, but you will need more comments in longer and more detailed sections of code. Your code should be properly indented, you should use good variable names, you should follow naming conventions, you should avoid magic numbers, etc. Everything that has been mentioned as a "best practice" in our lectures should be followed, or you risk losing points on your assignment

Your code should be easy to read. Ask yourself, would someone with no knowledge of this assignment be able to tell what is happening in the program from quickly reading my code? If not, then you need more comments.

Your code should follow the Model-View-Controller architectural pattern. The files are already structured to do this with a model, view and controller classes for the game setup process and for the game itself.

Files:

`ConnectXApp.java`

This class is the entry point of our Extended Connect X program. It calls the `GameSetupScreen` and `GameSetupController` class and turns the control of the program over to the event dispatch thread which will wait for an event to occur. **You do not need to make any changes to this file.**

`SetupView.java`

This class contains the code to create and layout the GUI for the setup screen. It also is the observer of the submit button. When someone clicks on submit, the `actionPerformed` method is called, which then calls the controller object.

You do not need to make any changes to this file, but it is a good example of a Java Swing GUI. Read through it to gain more understanding of how Java Swing works.

`SetupController.java`

This class is the controller for our setup screen. The `processButtonClick` method is called by `GameSetupView` when someone clicks on the submit button. It is passed in the rows, cols, players and the number to win by the view, but it still needs to validate that input. If there are any errors it can use the `displayError` method in the `GameSetupView` class to inform the player of the error, then wait for them to fix it and resubmit.

If there are no errors, it will create a new `IGameBoard` object (the implementation will depend on the size of the game board) to serve as the model, and the `ConnectXController` and `ConnectXView`. Control is then passed over the event dispatch thread that will wait for an event to occur. **Note:** this class expects the constructor for your implementations of `IGameBoard` to take its arguments in the order of rows, columns, and number to win.

Note: this class has stricter limitations on the size of the board than `IGameBoard` does. This is because the GUI is a little more limited about what it can fit on the screen. This is perfectly fine, and we do not need to change `IGameBoard` to meet these new limitations. Since the max size of the GUI is less than the max size of the `IGameBoard` we already know we will be meeting the constraints of `IGameBoard`.

`ConnectXView.java`

This class is the view of our Connect X game. Our view has a message area, and a list of buttons. The buttons will be arranges in a `ROWS_IN_BUTTON_PANEL` x `COLUMNS_IN_BUTTON_PANEL` Grid. Players will use another set of buttons above the grid to select the column to place the marker in. All events will be passed to the controller by calling the `processButtonClick` method.

This class also provides many methods that your code will need to call to interact with the user. It is important that you fully understand this class to be able to complete this assignment. **You do not need to make any changes to this file.**

`ConnectXController.java`

This class is the controller for the game. It uses the `ConnectXView` object to interact with the user and the `IGameBoard` object as its model to track what's happening in the game. This is the file you will need to complete.

You may need to add private data variables, and make changes to the constructor. You will need to complete the `processButtonClick` method, which handles everything about the process when a user clicks on a column to place a token, including input validation. You will need to use the methods in the `ConnectXView` class and the methods in the `IGameBoard` object to accomplish this.

A `newGame` method is provided to send the program back to the setup stage. You do not need to make any changes to this method, but you will need to call it. After someone wins the game, you will display to the players who won, and tell them to click any button to start a new game. If they click a button, you will then start a new game. Remember that clicking the button will call `processButtonClick` and return to the top of the method. The same should happen for a tie game.

While there will be some similarities to the `GameScreen` class in previous submissions, there will also be some key differences. Remember that in our controller we are now waiting for events to occur and then we respond to them (event-driven programming), while in previous assignments we had control over when the user could accomplish certain tasks (procedure-driven programming). Instead of having large loops to keep the game going and enforce the rules of our game, we will now be waiting for a user to click on a button, which will call the `processButtonClick` method in our controller.

Make sure your game meets these requirements:

a. The screen should say who's turn it is
b. When the user clicks a button to select a space, the text of the button should change to the players character (if the space was available)
c. The screen should give an error message if the user selects a spot that is not available
d. The game should alternate between players (unless the player picks an invalid space)
e. The game should check for a winner, and print a message if a player wins
f. The game should check for a tie game, and print a message if it occurs.
g. In the event of a tie game or a win, the next time the player clicks a button should start a new game.

The Program Report

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

<u>Requirements Analysis</u>

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. Your user stories should not have specified using a command line interface, so they should be largely unchanged. **Your non-functional requirements may be affected.**

Design

You need to provide a class diagram for the `ConnectXController` class. You also need to provide an activity diagram for the `processButtonClick` method and any methods you add to the `ConnectXController` class. You do not need to create an activity diagram for the `newGame` method or Constructor. You do not need to create class diagrams for the other classes. You do need to submit the diagrams for `IGameBoard` and its implementations from previous project assignments.

Testing

Your report should include the test cases created for project 4, but you do not need to create any new test cases for this assignment.

Deployment

You will need to update your deployment instructions that indicates that this is an IntelliJ project. We will not be able to run this program from our SoC Unix command line since it uses a GUI, so you do not need to provide a `makefile`. See the **Submission** section below for more details.

General Notes:

- Remember, this class is about more than just functioning code. Your design and quality of code is very important. Remember the best practices we are discussing in class. Your code should be easy to change and maintain. You should not be using magic numbers. You should be considering Separation of Concerns, Information Hiding, Encapsulation, and Programming to the interface when designing your code. You should also be following the idea and rules of design by contract. You should be following the Model-View-Controller architectural pattern.
- You class files should be located in `cpsc2150.extendedConnectX.controllers`, `cpsc2150.extendedConnectX.models` or `cpsc2150.extendedConnectX.views` packages
- Your code should be well formatted and consistently formatted. This includes things like good variable names and consistent indenting style.
- You should not have any dead code in your program. Dead code is commented out code.
- 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.
- The setup of the game no longer allows plays to pick their own marker. You can assign markers for them, knowing you will have at most 10 players in your controller class. Pick letters that look distinct (e.g., `X`, `O`, `A`, `M`…) and not similar letters that could be confused with each other when scanning the board (e.g. `B`, `R`, `P`, `D`…). These markers can be hard coded into your controller class.
- The provided code only requires a few minor changes to compile, so I expect every submission to compile. **Any submissions that do not compile will receive a 0.**
- Do not make changes to files you are not supposed to change
- If your previous code was incorrect, you will have issues in this assignment as well. Make sure to make those corrections for this submission

- We will not use some of the functionality that we provided in previous projects (such as the `toString` method). That is ok. You should not remove that method from your code, just ignore it.
- While this assignment should not take nearly as much time to complete as previous assignments, you will have less time to complete it, and it does involve material that we did not get to devote much lecture time to. Do not wait until the last minute to start this assignment.
- Make sure to test your assignment thoroughly by playing several games
- Make sure to submit the entire project directory so your assignment can be graded.

Tips and Reminders

- Do your design work before you write any code. It will help you work through the logic and help you avoid writing code just to delete it later.
- When writing your code, try to consider how things may change in the future. How can you design your code now to be prepared for future changes and requirements?

Submission

You will submit your program on Gradescope, but since we are working with GUI's there are a few changes.

1. You do not need your code to run on SoC Unix
2. You do not need to provide a `makefile`
3. You will need to zip up the whole IntelliJ project (like we did in the GUI lab). This code will not run on SoC Unix, so the TA will need to be able to unzip your file and open the project in IntelliJ to test it.
4. You will need to submit your program report as well, so place it in your folder with the IntelliJ project.
5. **If the TA is unable to open up your project in IntelliJ because you did not submit the entire project directory, they may not be able to grade your assignment which may result in a grade of 0.**

Your assignment is due at 11:59 pm. 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.**

Academic Dishonesty

This is an INDIVIDUAL assignment. You should not collaborate with others on this assignment. See the syllabus for further details on academic dishonesty.

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.

Checklist

Use this checklist to ensure you are completing your assignment. **Note:** this list does not cover all possible reasons you could miss points, but should cover a majority of them. I am not intentionally leaving anything out, but I am constantly surprised by some of the submissions we see that are wildly off the mark. For example, I am not listing "Does my program play Connect X?" but that does not mean that if you turn in a program that plays Checkers you can say "But it wasn't on the checklist!" The only complete list that would guarantee that no points would be lost would be an incredibly detailed and complete set of instructions that told you exactly what code to type, which wouldn't be much of an assignment.

- Can I change the size of my game board?
- Can I set the number in a row needed to win?
- Can I change the number of players in my game?
- Did I create an interface for my game board?
- Did I provide my interface specification?
- Did I move my contracts from my `GameBoard` class to my interface?
- Did I implement all secondary methods as default methods in my interface?
- Do I have both implementations of my game board?
- Does my game allow for players to play again?
- When players start a new game can they change the size of the board or the number to win?
- When my players start a new game, can they change the number of players?
- Does my game behave exactly the same for both implementations?
- Did I correct my contracts?
- Are my implementations efficient?
- Does my game take turns with the players?
- Does my game correctly identify wins?
- Does my game correctly identify tie games?
- Does my game run without crashing?
- Does my game validate user input?
- Did I protect my data by keeping it private, except for `public static final` variables?
- Did I encapsulate my data and functionality and include them in the correct classes?
- Did I follow Design-By-Contract?
- Did I provide contracts for my methods?
- Did I provide correspondences to tie my private data representation to my interface?
- Did I follow programming to the Interface?
- Did I comment my code?
- Did I avoid using magic numbers?
- Did I use good variable names?
- Did I follow best practices?
- Did I remove "Dead Code." Dead Code is old code that is commented out.
- Did I make any additional helper functions I created private?
- Did I use the `static` keyword correctly?
- Does my game use the provided Graphical User Interface?
- Does my program follow the Model View Controller Architectural Pattern?
- Does my game remove old error messages after the player has corrected the issue?
- Did I generate Javadoc for my project?