Homework 2

CSCI 2300

Understanding the code [10 points]

Review the code in the hw2 directory of your csci2300 git repositories. Compile and run this code. Examine all the classes in this code.

Answer the following questions about the code:

1. We know that every class implicitly extends the java.lang.Object class. Aside from that, where is inheritance used in this code? Specify all parent classes and subclasses. For example, if class C extends class B and class B extends class A, you would write:
   A ← B ← C
2. Board class is declared as abstract. What does that mean?
3. In the constructor of `TicTacToeBoard` class, what does the following line of code do?
   `super(3, 3);`
4. Recall that a *public interface* of a class is a set of methods and variables that can be accessed outside of that class via an object of that class. Specify the public interface of the `TicTacToeBoardPanel` class.

Type up your answers to these questions and turn them in.

Improve the code

While the provided code works, it could be improved. Make the improvements specified in this section.

Improve cohesion via inheritance [20 points]

Recall, that in homework 1, our tic-tac-toe game used command line terminal for input. That code had components similar to those in homework 2. Let's focus on the `ScoreBoardPanel` class. `ScoreBoardPanel` class was created by adding code to `ScoreBoard` class of homework 1 and renaming the class. Since we renamed this class, our solution to homework 1 will no longer compile.

One way to overcome this problem is to keep the old name, `ScoreBoard`. However, while that would work, the Java SWING component of the new score board is unnecessary for the terminal version of the game.

Another way to overcome this problem is to use inheritance. We could keep old `ScoreBoard` class, and create `ScoreBoardPanel` as a subclass of `ScoreBoard`. This way, the terminal version of the game will work and will not include any unnecessary components. The GUI version of the game will also work.

Modify `ScoreBoardPanel` class by splitting it into `ScoreBoard` parent class and `ScoreBoardPanel` subclass. Make decisions on which methods should reside in which class. Ensure that the GUI version of the game still works. Note, how this improves code cohesion, because the logic of keeping score and announcing the winner is contained in the `ScoreBoard` class and the logic of presenting the score on the GUI application is now contained in the `ScoreBoardPanel` class.

No responsibility sharing and code cohesion [20 points]

Look at the `ScoreBoardPanel` class (or the modified version `ScoreBoard` and its subclass `ScoreBoardPanel`). There are methods that have more than one responsibility: `announceWinner()` increments the number of wins and announces the winner. The `announceTie()` method increments the number of ties and announces the tie. This is a sign of **low cohesion.** Additionally, `ScoreBoardPanel` class shares the responsibility of announcing the winner with the `AnnouncementPanel` class. This violates the 'single responsibility principle' of object oriented design.

Suppose we decided that the `AnnouncementPanel` class should be the one responsible for announcing the winner and `ScoreBoardPanel` class should be responsible for keeping track of the number of wins per player and the number of ties. Modify the code in the `ScoreBoard`/`ScoreBoardPanel` class such that it is consistent with the score board's responsibility. Note, that since `ScoreBoard/ScoreBoardPanel` will no longer be announcing winners and ties, the methods in this class will need to be renamed, to be consistent with what those methods do. After these code changes, you will need to modify the code that uses `ScoreBoardPanel` class to ensure the application still compiles and works.

Add New Code

Different player types [50 points]

Consider the following requirements of the TicTacToe game:

R1. Two human players should be able to play a game of TicTacToe using a terminal

R2. Two human players should be able to play a game of TicTacToe using GUI.

R3. A human player should be able to play a game of TicTacToe versus a computer player using a terminal.

R4. A human player should be able to play a game of TicTacToe versus a computer player using GUI.

We have implemented requirements R1 and R2 with homework 1 and homework 2. Let's extend our design, to achieve requirements R3 and R4.

First, let's note that we have three types of players here:

- Human player using a terminal
- Human player using GUI
- Computer player

Since the responsibility of a player is to 'take a turn', we should be able to generalize this using inheritance. Add an abstract `Player` class and three classes that are subclasses of `Player`: `TerminalPlayer`, `GUIPlayer`, and `ComputerPlayer`. Decide which method(s) in the Player class should be abstract. Make sure to implement the abstract methods of the `Player` class in the derived classes. (Note, you don't have to make `ComputerPlayer` be intelligent, it can pick board positions at random or in order. If you need to add methods to the `Board` class to give `ComputerPlayer` visibility to the board, you may do so). Hint: you may need to override the

`setPiece()` method of the `Board` class in the `TicTacToeBoardGUI` class. Make sure there are no redundancies in your code – the same functionality executed multiple times.

To ensure that the application still works, adjust `TicTacToeGUI` class to use `GUIPlayer`.

Playing against ComputerPlayer [10 points]

Write `TicTacToeAI` class that uses `TicTacToeGUI` class to have a human play a game of tic tac toe against a computer. Your `TicTacToeAI` class should contain only one method:

```
public static main(String [] args)
```

This method will instantiate a `GUIPlayer` object, a `ComputerPlayer` object and a `TicTacToeGUI` object and call `start()` method on the `TicTacToeGUI` object.

You may need to adjust the `actionPerformed()` method of `GUIGameController` (which is a class nested in the `TicTacToeGUI` class) to ensure that the computer player gets a turn to play. Hint: call `takeTurn()` method after switching the current player and implement the `takeTurn()` method of the `GUIPlayer` to do nothing.

Conceptual Integrity [10 points]

It is important to keep code consistent. We now have a `TicTacToeGUI` class that implements the GUI logic and has the main method and a `TicTacToeAI` class that allows a human to play against the computer using `TicTacToeGUI` class. To make this code more consistent, remove the `main()` method from the `TicTacToeGUI` class and implement a new class `TicTacToeHuman`. The `TicTacToeHuman` class will have only method:

```
public static main(String [] args)
```

This method will instantiate two `GUIPlayer` objects and a `TicTacToeGUI` object and will call `start()` method on the `TicTacToeGUI` object.

Submitting your code

Merge your code to the master branch of your csci2300 git repository and push the code to the git server. Ensure that this worked by going to git.cs.slu.edu and checking if you see your updates via the web interface.

Grading: points for each section are listed in the assignment. Do not turn in code that does not compile. If you turn in code that does not compile you will get **at most** 50% of the points for the completed sections (even if a given section was completed correctly and some other section causes compiler errors).

A good way to manage this is with git branches. Create a new git branch for each section of the assignment. Do the work specified in that section. Once everything works, merge your code to the master branch and push it. Then move on to the next section. If you have trouble with a given section, you can skip it by checking out the master branch (which should contain working code) and creating a new dev branch. This will ensure that master does not get broken.