MVC Design: Tic Tac Toe
5004 Object Oriented Design

1. Goals

- Explore MVC design
- To create an application using MVC design
- To avoid code reuse

2. In Recitation:

This is likely the most complex application you have created so far. I'd like you to do the following in recitation:

1. Have a planning session with your TA or a peer group
2. Create the framework for your application
3. Review all instructions and ask your TA any questions you have

3. Instructions

Part 1 The Model:

The purpose of this exercise is to give you practice with implementing the Model component of the Model-View-Controller design pattern. You'll be depending less on creating a complete application and focus more on a set of constraints that have been presented to you already.

In the starter code, you are given an interface representing a game of Tic Tac Toe; your task is to implement the TicTacToe interface.

1. Review the MVC design for yourself and make sure you understand what you are trying to implement.
2. Study the existing interface so you understand the overall goals of what you are creating.
3. Begin creating your model by implementing your interface.

You will need to define an enum Player, representing the players (X and O), with a `toString()` method that returns "`X`" and "`O`" accordingly.

You will need to implement the public class named `TicTacToeModel`, with a single public constructor that takes no arguments.

The class definition, with a `toString()` implementation to help with debugging, is provided to you in the starter code.

You will fill in the fields and remaining method definitions as appropriate. You may also define other classes as needed.

The game grid cells are numbered by row and column starting from 0. For example, the upper left position is row 0, column 0 (or `[0][0]` in the 2D array returned by `getBoard()`), the upper middle position is row 0, column 1 (`[0][1]`), the lower right is `[2][2]`.

You might want to research how to create arrays and mult-dimensional arrays in Java.

Testing

We have supplied you with some basic JUnit tests as part of the starter code. Use these to verify that your implementation is correct, and write additional tests of your own as needed.

Notes to Keep in Mind

- Avoid duplicating code as much as possible. Consider using non-public methods as means of creating reusable pieces of functionality.
- Be sure to use access modifiers, `private` and `public`, as well as final, appropriately.
- In your getters, be careful to return a **copy** of, and not a direct reference to, any mutable internal state in your model.

Starter Files:

https://www.dropbox.com/s/407yif6l0zzf4tq/TicTacToe.java?dl=0
https://www.dropbox.com/s/euolh01mbny4n22/TicTacToeModel.java?dl=0
https://www.dropbox.com/s/71is8k6309f3l09/TicTacToeModelTest.java?dl=0

Part 2: The Controller

The purpose of this exercise is to give you practice with implementing the Controller component of the Model-View-Controller design pattern, by means of a textual, console-based controller.

In the starter code, you are given an interface representing a controller for Tic Tac Toe, with a single method, `playGame()`. Your task is to implement the `TicTacToeController` interface. Put your new controller code in the same package alongside your Tic Tac Toe model from the previous exercise as it will depend on the model. You are also given a class `Main` with a `main` method that will allow you to test your game interactively.

You will need to create one additional class: a public class named `TicTacToeConsoleController` that implements `TicTacToeController`, with a single public constructor that takes two arguments, a `Readable` and an `Appendable` (in that order). You will fill in the fields and the method definitions as appropriate. You may also define other classes at your option as needed.

The controller will output game state and prompts to the `Appendable`, and read inputs from the `Readable` corresponding to user moves.

The `append()` method on `Appendable` throws a checked exception, `IOException`. Your `playGame()` method should not throw this exception. If it occurs, your `playGame()` should catch it and throw an `IllegalStateException`.

A single move consists of two numbers, specifying the row and column of the intended move position. Board positions for these moves are numbered from 1. For example, to mark **X** in the upper left cell, the user would enter "`1 1`" at the first prompt. To mark **O** in the upper right cell on the second move, the user would enter "`1 3`". To quit a game in progress, the user can enter `q` or `Q` at any time.

The game state is the output of the model's `toString()` method, followed by a carriage return (`\n`). The move prompt is

```
"Enter a move for " + model.getTurn().toString() + ":\n"
```

(where model is an instance of your Tic Tac Toe Model).

If a non-integer value is entered, it should be rejected with an error message. If an invalid move is entered, namely, two valid integers, but the proposed move was deemed invalid by the model, the controller should give an error message. The message text is up to you, but should end with a carriage return.

At the end of the game, the controller should output, in order on separate lines:

- A final game state
- `"Game is over!"` followed by `"X wins."` or `"O wins."` or `"Tie game."` depending on the outcome

If the user quits, the controller should output

```
"Game quit! Ending game state:\n" + model.toString() +
"\n"
```

and end the `playGame()` method.

Testing

We have supplied you with some basic JUnit tests as part of the starter code. Use these to verify that your implementation is correct. **Write additional tests of your own:** Some of the additional cases you should consider are listed as comments in the test class you are given.

Notes to Keep in Mind

- You will likely need a `while()` loop; be aware that you can use the `break` statement to break out of a loop prematurely (before the loop condition is `false`).

- You will need to use the built-in `Scanner` class. See the lecture notes examples and explore the Oracle JavaDoc for this class for more information on how to use it.

- Avoid duplicating code as much as possible. Consider using non-public methods as means of creating reusable pieces of functionality.
- Be sure to use access modifiers, `private` and `public`, as well as `final`, appropriately.
- Include JavaDoc for your classes and constructors as appropriate. You do not need to repeat JavaDoc already existing in a superclass or interface when you override a method.

To Turn In

Submit your zip containing only your src and test directories as a zip file in Canvas.

Files:

- https://www.dropbox.com/s/0gpla0xkclnjfhh/Main.java?dl=0
- https://www.dropbox.com/s/o61w040sfgs8gv4/TicTacToeController.java?dl=0
- https://www.dropbox.com/s/olnbk38vpw20053/TicTacToeControllerTest.java?dl=0
- https://www.dropbox.com/s/5nktrv5vh0eilje/FailingAppendable.java?dl=0

4. Extensions:

You can get 90% by completing all of the above requests, but to push up to 100%, you'll need to go above and beyond what's asked for. I would like you to add a few elements of your own. The number of points you get for your element depends on its complexity. Make sure to include a text file called Extensions.txt explaining the extensions you added to this assignment.

- Go above and beyond with testing
- Creating a driver to test your implementation
- Add additional functionality
- Go above and beyond with documentation
- Create a view component

5. Report:

Each assignment must include a short report. The generation of this report should take you no more than 15 minutes. This gives you a chance to reflect back on what you learned and it makes grading easier on your grader. For this report, I want the following sections:

1. Reflection (*What did you learn?)*
2. How does this design encourage future growth?
3. Extensions (*What extensions are you requesting?)*
4. Grading Statement (*Based on the rubric, what grade do you feel you deserve? Be honest.)*

Rubric

	Possible	**Given**
Part 1 : The Model		
As requested	4	0
testMove	4	0
testHorizontalWin	4	0
testDiagonalWin	4	0
testInvalidMove	4	0
testMoveAttemptAfterGameover	4	0
testCatsGame	4	0
testInvalidGetMarkAtRow	4	0
testInvalidGetMarkAtCol	4	0
testGetBoard	4	0
Part 2 : The Controller		
testSingleValidMove	4	0
testBogusInputAsRow	4	0
testTieGame	4	0
testFailingAppendable	4	0
Overall		
Code Quality	10	0
Report	4	0
Application meets description	10	0
Code duplication avoided	5	0
Data elements protected	5	0
Not included in total possible:		
Driver not included as requested	-100	0
Does not compile	-100	0
Extensions (Not calculated without report)	10	0
Late penalty	-20	0
Creative or went above and beyond	10	0
Code contains warnings	-20	0

TOTAL POINTS POSSIBLE out of 100	90	0