CMPU-102 – Data Structures and Algorithms (S22)

Hannah Gommerstadt and Rui Meireles

Homework #2 – Tic-tac-toe

Due Fri April 1st 11:59pm, Checkpoint on Fri March 4th 11:59pm

Introduction

In this homework you will implement a Java version of the popular game Tic-tac-toe. This handout assumes familiarity with the game. If this is not the case, please consult Wikipedia before continuing. You can also play it online, e.g. at https://playtictactoe.org.

You shall implement a one-player version of Tic-tac-toe. The second-player will be controlled by the computer itself. No advanced artificial intelligence will be required, the computer will play randomly.

Functional specification

The program must function as specified by the sequence diagram of Figure 1.

The traditional version of Tic-tac-toe is played on a 3×3 board. Since this version of the game is not very challenging, your program shall support a generalized variant that can be played on larger boards. For this purpose the program shall take in a positive integer argument n representing the size of the board, which will be $n \times n$. The game should support board sizes between 1×1 and 9×9. The rules for this generalized Tic-tac-toe are the same as for the regular version: the player who completes a line, column or diagonal first, wins. If no player manages to win before the board becomes full, a tie is declared.

This shall be a text-based game. The following format is to be used for the game board:

```
   1   2   3
A     |   | X
  ---|---|---
B     | O |
  ---|---|---
C     |   |
```

The human player's moves shall be represented on the game board by an uppercase `X`, and the computer's moves by an uppercase `O`.

Game board rows should be labeled using sequential letters, starting with `A`. Columns should be labeled with sequential integer numbers, starting with `1`.

The human player shall interact with the program via the keyboard. Moves shall be specified using a combination of two characters `rc`, where:

`r` is a letter representing a row on the game board;

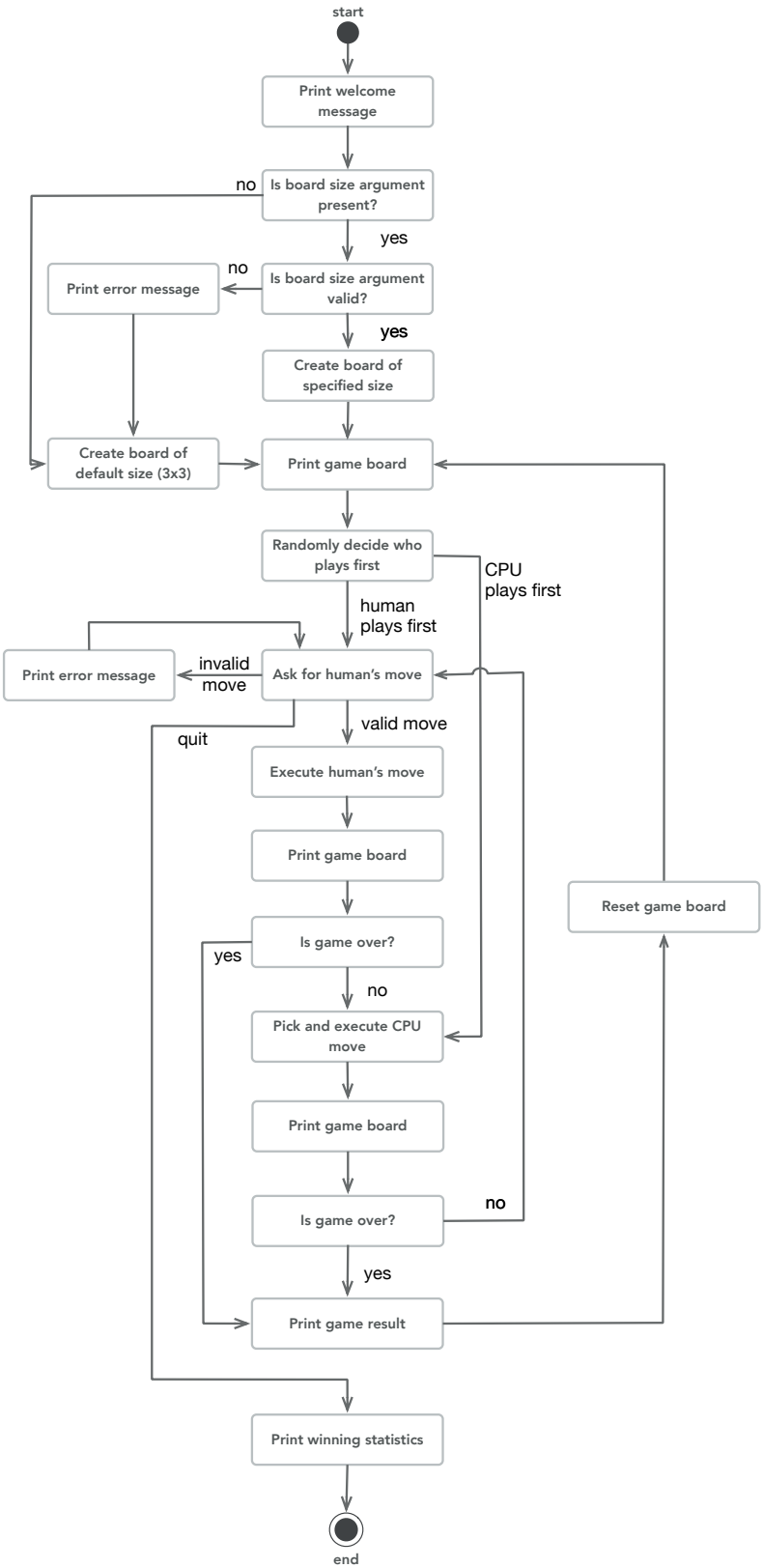

Figure 1: Tic-tac-toe sequence diagram.

c is a single-digit number representing a column on the game board.

There can be no spaces between the two characters. For instance, the moves yielding the game board depicted above would be `A3` and `B2`. Please allow the user to specify the row in a case-insensitive manner. I.e. both `A3` and `a3` should be accepted as referring to the 3rd column of the 1st row of the game board.

The program should be able to handle bad user commands gracefully without crashing or misbehaving, catching exceptions if necessary.

When a game of Tic-tac-toe is finished, a new one should start immediately after. The program should only terminate when the user specifies `quit` as its next move. This should also be a case-insensitive command.

When the player's quits, the total number of played games and his winning ratio should be printed on screen.

So you can better understand what the program's behavior should be, a reference implementation is provided (source code not included, naturally). To run it from the command line, type `java -jar tictactoe.jar <board-size>`, where `<board-size>` is an integer in $[1, 9]$ representing the desired board size. For example, running `java -jar tictactoe.jar 4` will have you play a Tic-tac-toe game with a 4×4 board.

Implementation guidelines

To guide you in your work, complete Javadoc documentation for the reference implementation is provided. Please study it carefully. You may choose to follow this reference design or create your own.

Regardless of how you code it, the program should always adhere to the functional specification described in the previous section and exemplified by the reference implementation.

The reference implementation has the following classes:

`Game` The class that represents the game state and workflow. This is where the `public void main(String[] args)` method is located. The main data structure in this program is the game board. `Game` uses a 2D-array of characters to represent it.

`GameStats` Stores statistics regarding the number of games won, lost and tied by the human player.

`Move` Represents a move in Tic-tac-toe, which is composed by a position on the game board (row and column). By aggregating row and column into a single data type, this simple class allows us to overcome one of Java's limitations: the fact that methods can only return one value.

`APlayer` Represents a generic Tic-tac-toe player. Its main responsibility is to provide structure to the concrete player classes.

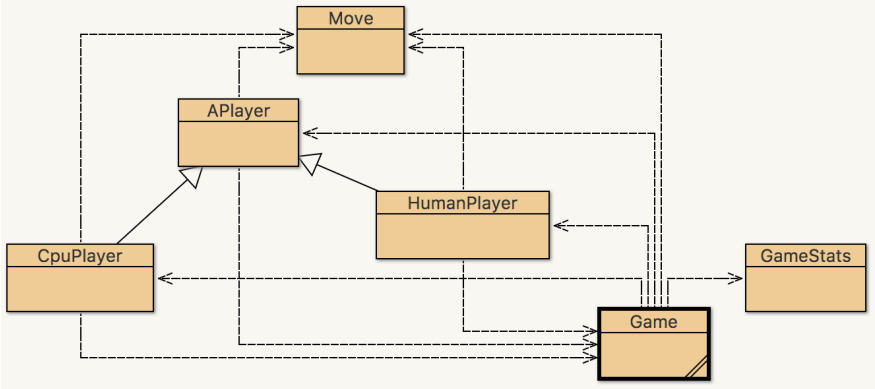

Figure 2: Reference implementation class diagram

`HumanPlayer` A concrete Tic-tac-toe player that picks moves based on what it is told by a human user. Human-computer interaction is performed through the keyboard.

`CpuPlayer` A concrete Tic-tac-toe player that picks moves uniformly at random from the still unoccupied positions on the game board.

Figure 2 depicts the inheritance and use relationships between these classes, as seen in BlueJ. The Javadoc documentation includes complete descriptions of all their fields and methods.

As a general rule, we recommend you start building from simpler to more complex classes, testing as you go along. You can use static helper methods to set up controlled game scenarios to test your methods individually.

Additionally, here is a list of some useful methods to handle user input:

`boolean equals(Object anObject)` A method of the `String` class that returns `true` if and only if the argument is not `null` and is a `String` object that represents the same sequence of characters as the `String` object it's called on.

`String toUpperCase()` A method of the `String` class that returns a new `String` that is a version of the `String` it's called on where all lowercase letters have been replaced by their uppercase counterparts.

`char charAt(int index)` A method of the `String` class that returns a `char` representation of the character at position `index` in the `String` it's called on.

Additional notes and tips

- **To make sure that you are making progress on the assignment, there will be a checkpoint on Friday March 4th 11:59pm. You will need to show a coach that you can print the board and put moves on the board before then. You do not need to upload code to Moodle for the checkpoint.**

- You are required to document your code using Javadoc.
- Remember that, in Java, `char` is a numeric data type. You can do arithmetic with characters. Since letters and digits were assigned sequential and increasing numeric values, `'B'-'A'` will evaluate to `1`, and `'3'-'1'` will evaluate to `2`. You can use the command `man ascii` on a Unix shell, or visit http://www.asciitable.com, to learn more about the standard mapping between characters and their numeric values.
- You may elect to write a program that only works for 3×3 game boards. This will cause you to miss 25% of the total amount of points available.
- You may elect to write a more sophisticated playing strategy for the computer-controlled player for extra credit (awarded in accordance with its quality). In general terms, an intelligent computer player should, from the moves available, pick one that maximizes its chances of winning the game.
- We recommend the following development sequence:

  1. Print the game board. This will be similar to how you printed matrices in the Matrix calculator lab.
  2. Create the player classes. Remember that, as mentioned above, you can do arithmetic with characters.
  3. Write code to play a single game, alternating between players.
  4. Write code to perform game-over detection.
  5. Write code to play multiple consecutive games and collect performance statistics regarding wins, losses, and ties.
  6. Wrap up any loose ends.

- Perform incremental tests after each adding each feature. You want to avoid a situation where you have a lot of code written but untested, as it makes for much harder debugging and increases the likelihood of having to scrap large amounts of work. Remember it is always better to hand in a feature-poor program that works well than a feature-rich program that doesn't work at all.

  You can leverage static methods to help you test features. For example, to test the game-over detection, write a method that creates a game instance, programmatically places a relevant set of pieces on the board, calls the end game detection method, and finally checks whether the result is the expected one. Do this for all relevant board configurations (e.g. unfinished game, column win, row win, diagonal win, tie).

  At the end, make sure to test the integration of all individual components. The idea is simple: you are making a product, which is itself made up to multiple pieces. First, make sure each individual piece is good. Then, make sure they interact well with each other.

- Use coaching and office hours wisely. First, try to work through problems yourself, which will help you develop your analytical thinking skills. But, if you find yourself stuck after that, do not hesitate in reaching out to a coach or instructor. There is no stigma or penalty, we're here to help you!

Submission

Once you are satisfied with your program, compress (tar.gz or zip format only, please) and submit your code for evaluation through the course's Moodle page.