Ultimate Tic-Tac-Toe (18 points + 3 bonus points)

In this project, your task is to implement the game Ultimate Tic-Tac-Toe (UTTT). Ultimate Tic-Tac-Toe is a more complex adaption of the classic Tic-Tac-Toe game. In case you are not familiar with the gameplay of UTTT, refer to section 1 describing the game rules.
This project is split split into two phases. In the first phase, you will implement tests that are intended to detect errors in faulty implementations. In the second phase, you will create your own game implementation. This structure aims to mimic a real-world scenario where you must create your own tests to validate your work. The submission of the tests (first phase) is due after the first week of this project. Note that the last push before the respective submission deadline is used for the evaluation of the respective part project evaluation.
UTTT is often commonly used as a convenient entry point to AI development, as it provides a balanced challenge that is harder than Connect 4 yet simpler understand than Chess. Thus, as a bonus exercise, you'll have the opportunity to implement an AI for both the classic version of Tic-Tac-Toe and Ultimate Tic-Tac-Toe.

To obtain the project, you can clone it into the folder `project4` via the command

```
git clone ssh://git@git.prog2.de:2222/project4/$NAME.git project4
```

where you have to replace $NAME with your CMS username. To work on this project, utilize Visual Studio Code (VSCode). Follow these steps:

1. Start VSCode.
2. Click on *File* > *Open Folder*.
3. Navigate to the directory where you cloned the project to, using the file explorer and click *Open*.
4. When prompted, select "Yes, I trust the authors."

In the project's first phase, running tests locally is not possible. However, when you open the project in VSCode, it recognizes the files as part of a Java project, and it immediately flags any semantic or syntactic errors that could prevent the project from compiling.
In the second phase, you can run your self-written tests on your own UTTT implementation using the Experiments tab (Erlenmeyer flask icon) in VSCode. To interactively test your full implementation using the game's GUI, use the run/debug tab of VSCode to run the `main` function of the project in `uttt.game.main`.

1 UTTT Gameplay

The board is composed of 9 Tic-Tac-Toe grids arranged in a 3x3 grid. The following is a brief description of the gameplay:

- The game starts with the first player placing their mark (X or O) in any cell of any smaller board.
- The move determines the board on which the next player will play. Specifically, if the first player places their mark in the top right cell of a small board, the next player must play on the top right small board.
- This rule continues, with the position of a move within a small board dictating where the next move must be made within the larger board.
- To win a small board, a player must get three of their marks in a row, either horizontally, vertically, or diagonally, as in traditional tic tac toe.
- The larger game is won by winning three small boards in a row, again either horizontally, vertically, or diagonally.

- If a player is supposed to make a move on a small board that is already won or drawn (entirely filled), they may play on any board that is not won or drawn.
- The game ends when a player wins the larger board or there are no legal moves left, in which case the game is a draw.

Figure 1 illustrates three sample UTT gameplay moves.

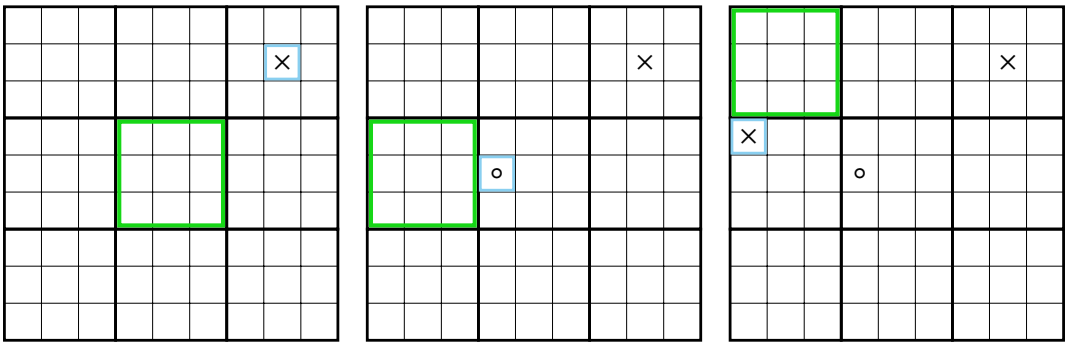

Figure 1: Three possible moves after the start of the UTTT game. Blue indicates that the according mark was just placed. Green indicates the grid in that the next mark needs to be placed.

2 Project Structure

While you will be responsible for the bulk of the implementation, the general structure is predefined through interfaces. This facilitates both our testing of your implementation and your testing of any implementations provided. We utilize the Factory Pattern for this purpose. In brief, the Factory Pattern is a design pattern that allows creating objects without exposing the implementation. For this project, you can define your implementations in `UTTTFactory.java`. This is also where you can query the factory to obtain instances for testing (potentially faulty) implementations. The server will potentially replace the factory file to run your tests on different implementations.

The project's directory structure is organized, into three separate packages. Respectively representing game logic (`uttt.game`), utility classes (`uttt.utils`) and the user interface (`uttt.ui`). The game logic package includes interfaces for the game board, player moves, simulation, and interaction. The utility package contains classes that represent game symbols and moves. The user interface package handles the graphical representation. The user interface is a convenient and fun way to test your implementation. You can ignore these classes for your implementation and tests.

The functionality you have to implement and test is contained in the game logic package (`uttt.game`). In the following you can find a brief description of each interface. More detailed information can be found in the project documentation. I.e. in the comments above the respective interfaces and their functions.

MarkInterface The `MarkInterface` describes the marks that are placed by the players on the individual boards. Each mark consists of the symbol of the player who placed it, as well as its position on the corresponding local board, as shown in Figure 2 as an example of how to choose the indices.

BoardInterface The `BoardInterface` describes the structure of a single Tic-Tac-Toe field in the game. The marks (`Mark`) of the individual players are placed on the boards. Before placing a mark, it is checked whether it is a valid move.

SimulatorInterface The implementation of the `SimulatorInterface` should provide the game logic to run a complete game. It determines whether the players are playing on the correct board and when a player has won or the game ends in a draw.

PlayerInterface The `PlayerInterface` forwards the player's moves to the simulator. It can forward moves from the UI if a human is playing, as well as moves from an AI.

GameRunnerInterface The `GameRunnerInterface` is a wrapper class to share the game run functionality between `BoardInterface` and `SimulatorInterface`. You are not required required to test the `run` functions. Though, you need to implement the `run` function of the SimulatorInterface in your own implementation. The `run` function in the `BoardInterface` may be used for assistance with the bonus exercise.

UTTTFactory The `UTTTFactory` is used to link and make the individual implementation classes interchangeable. Instances of the implementations are created and returned here in order to build a functional game in the Main class.

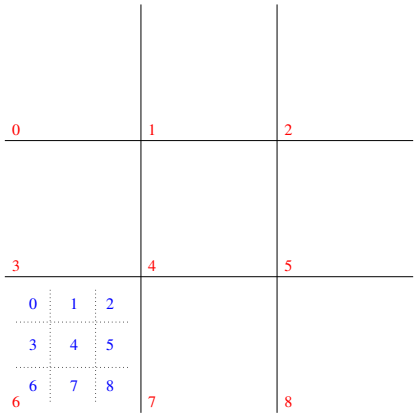

Figure 2: The red numbers represent the board indices. The blue numbers represent the mark indices within a board.

Java aims to provide numerous safety assurances. In line with this philosophy, many methods in the project are declared with the addition `throws IllegalArgumentException`. This indicates that the method aims to enforce these safety assurances by expecting valid input. If the method is called with invalid arguments, an IllegalArgumentException is thrown.
Consider the input restriction similar to a program precondition, as discussed in the chapter on formal correctness. The goal here is to enforce that the program always results in an abort state if the precondition is not fulfilled.

3 Assignments

3.1 Phase 1: Test Creation (9 points, Submission 13.06.2023 23:59)

In this part of the project, you need to implement JUnit tests. The tests should check implementations according to the description in the project documentation. Therefore, we will execute your tests with both faulty implementations and a correct implementations of the game logic interfaces in the project. For each faulty implementation, we expect at least one of your tests to fail, while passing successfully on the correct implementation.

Technical Notes

- You need to test all methods of the given interfaces `MarkInterface`, `BoardInterface` and `SimulatorInterface`, except for the method `run` from the `SimulatorInterface` and `BoardInterface`.
- For methods declared with `throws IllegalArgumentException`, you should test that an Exception is thrown on invalid input.
- All tests that should be used for evaluating this task must be implemented in the Java package `uttt.tests`.
- It doesn't matter whether you use multiple classes for this purpose or not. You can find a small example test in the class `SimpleTest`. If you want to implement your own helper methods or classes for your tests, they must also be implemented in the `uttt.tests` package.
- As a helpful tool, there is the test `prog2.tests.daily.TestsCorrect.all` that executes all of your tests once with the correct implementation and outputs the failing tests. In case of uncertainty, you can also use this test to understand how the reference implementation handles specific inputs or problems.

3.2 Phase 2: Implementation (9 points, Submission 20.06.2023 23:59)

In this part of the project, you are required to implement the game using the familiar interfaces. You should create your classes that implement the interfaces in the `src.uttt.game` package. Do not forget to adjust the `UTTTFactory.java` accordingly. Since you will be writing the tests yourself, there will be no public tests provided for your implementation.
Contrary to section 3.1, you must implement all the methods this time! Once you have implemented your own `UTTTFactory`, you can easily execute your tests in VSCode to test your own implementations of `ultimate-tic-tac-toe`.

3.3 Bonus: AI (3 bonus points, Submission 27.06.2023 23:59)

As additional challenge in this project, you have the opportunity to implement an artificial intelligence (AI), also referred to as a bonus or computer player in the code. The AI should implement the `PlayerInterface` and calculate some move given a game state. Understanding Tic-Tac-Toe is essential for playing Ultimate Tic-Tac-Toe, hence there are methods provided for computing moves for both game variants. While we don't test the run method in `BoardInterface`, you're encouraged to implement it as a practical way to test your TTT AI with the game GUI.

To grant you some additional time to for the AI implementation, we set the deadline to 27.06.2023 23:59, one week past the game logic implementation deadline. Please note that another project will also begin during this week. It is strongly advised to prioritize the new project over the AI implementation. However, if you find any additional time, you are welcome to accept the challenge and experiment with your AI implementation.

There are three test categories for your AI to compete in, each worth up to one point:

- Consistently achieve draws against a "perfect" Tic-Tac-Toe reference AI. (Note that winning against the reference AI isn't possible.)
- Win two out of three games against a random-move computer player in UTTT.
- Identify the shortest winning move sequence in a UTTT instance where a sequence of three moves or fewer is required to win.

Your AI implementation has 10 seconds to initialize, i.e. perform any computations during constructor call. Each move computation has a time limit of 3 seconds. Failure to return a move within the time limit will result in the AI losing the game. Scoring at least 2 out of 3 AI test points qualifies you for the end-of-semester AI tournament. (This is just for fun and bragging rights.)

For those unfamiliar with AI gameplaying, we recommend reading up on Alpha-Beta Pruning and Monte Carlo Tree Search, as they offer a good starting point for these tasks. You will find a plethora of resources if you search for these keywords (in combination with TTT or UTTT). AlphaZero may also be a good keyword for your research. However, we encourage creativity, so feel free to test out your own ideas.