Course Work

ITS1033 – Object Oriented Programming

Graduate Diploma in Software Engineering

Final Examination

Total Marks: 100

You **MUST** do this assignment individually and, unless otherwise specified, you **MUST** follow all the general instructions and regulations for assignments. **Please read this entire document before starting**.

Warm-up

In this assignment, your knowledge and development skills will be evaluated in following areas,

- Programming Fundamentals
- Matrices
- Concrete Classes, Abstract Classes, Interfaces, Leaf Classes, Nested Classes
- Enumerations
- Encapsulation, Inheritance, Polymorphism, Abstractions, Associations
- Hiding, Shadowing
- Java Bean Specification
- Lambdas and Method References
- Access Modifiers
- Recursions
- UML Class Diagrams

So take 20 minutes to recap and refresh your knowledge. This warm-up session is vital, so do not take it lightly.

Introduction

In this assignment, you will implement the complete logic behind the **Connect 4 Game** including the Artificial Intelligence part of the computer player. If you have never played the game before or if this is the first time you heard about this game, then take a couple of minutes to find out what Connect 4 Game is.

https://en.wikipedia.org/wiki/Connect_Four
Connect Four Gold Medal Match

Even though the objective of the game is to be the first to form a horizontal, vertical or **diagonal line** of four of connected pieces, **for the sake of the simplicity of the assignment we have ignored the diagonal line in this assignment.**

All the UI logic has been already implemented for you, so don't worry about the UI technology. **To successfully complete this assignment, you need zero knowledge and zero lines of code when it comes to the implemented UI technology (Java FX) of this assignment**. The objective of this assignment is not to find out how good you are with some sort of UI technology, but to find out how good you are in the Object-

oriented world. **If you follow the instructions right, if you implement the logic right, the game should also work fine**. **Let the UI work for you!**

If you are still unclear about the assignment's objective, please clarify your doubts here before proceeding.

In case if you wonder how the final game looks like after the completion of this assignment, **visit here**

✅ Prerequisites to complete this assignment

Make sure the following applications are already installed.

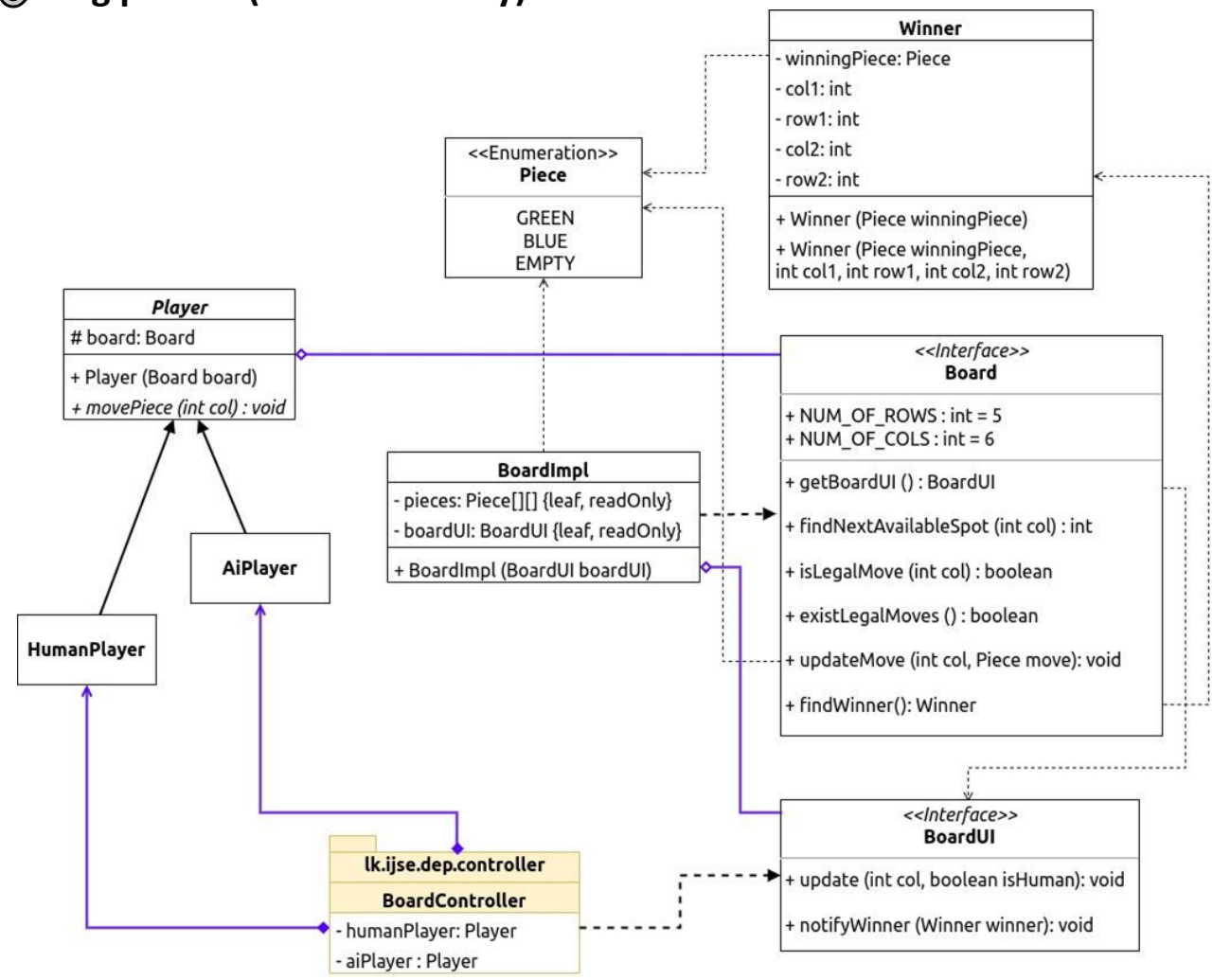
- Java 11 (Oracle JDK 11)
- Maven >= 3.8.1
- Git

🎮 Let's get started

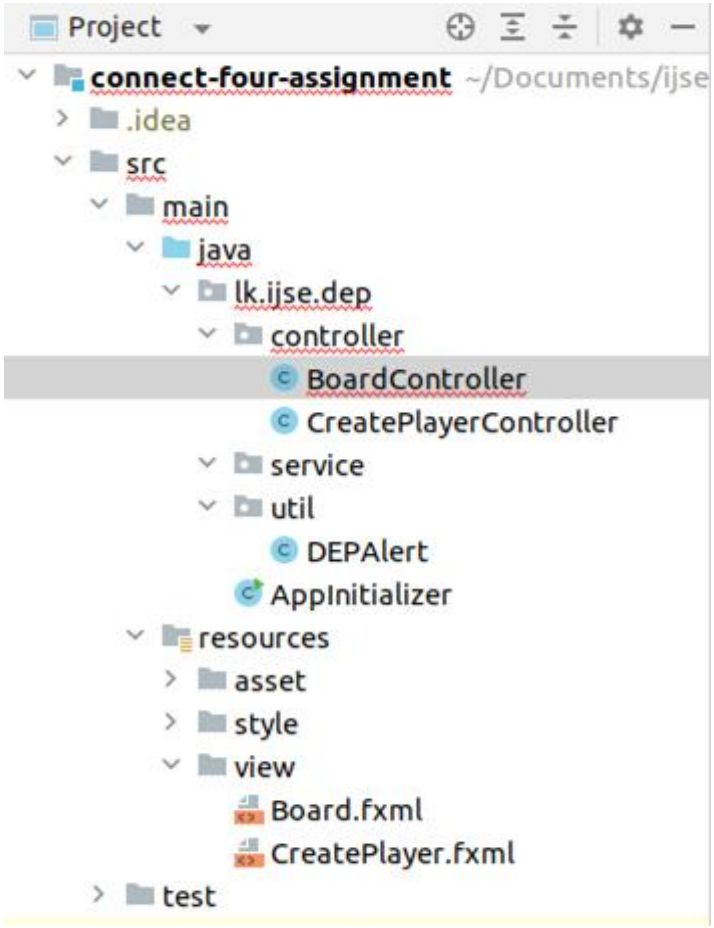
First of all visit the following link and follow the instructions there to get started

https://github.com/Ranjith-Suranga/connect-four-game-assignment.git

🥏 Big picture (Class Hierarchy)



Apart from the `lk.ijse.dep.controller.BoardController` class, **all other types** in the above diagram reside in the **`lk.ijse.dep.service`** package. Unless otherwise specified, you are only supposed to code in the **`lk.ijse.dep.service`** package. Graders will deduct the assignment marks significantly for deviations from the general instructions and regulations.



🎮 Step1: Getting class hierarchy ready (Marks: 30)

Once open the project via IntelliJ IDEA, you **SHOULD** see a similar project structure as in this image with plenty of compile time errors in the **`BoardController`** class. But don't worry about it now, because we are not going to run this application soon until we finish our class hierarchy correctly.

Once we start to create our class hierarchy step by step, the compile time errors will gradually fade away.

Do not worry about implementing the logic, instead focus on the structure of the type (Class, Interface, or Enum)

Tip: Start creating the class hierarchy that has least dependencies on other types.
Refer the above UML class diagram to identify members of respective types. Pay close attention to the symbols and arrows.

`lk.ijse.dep.service.Winner`

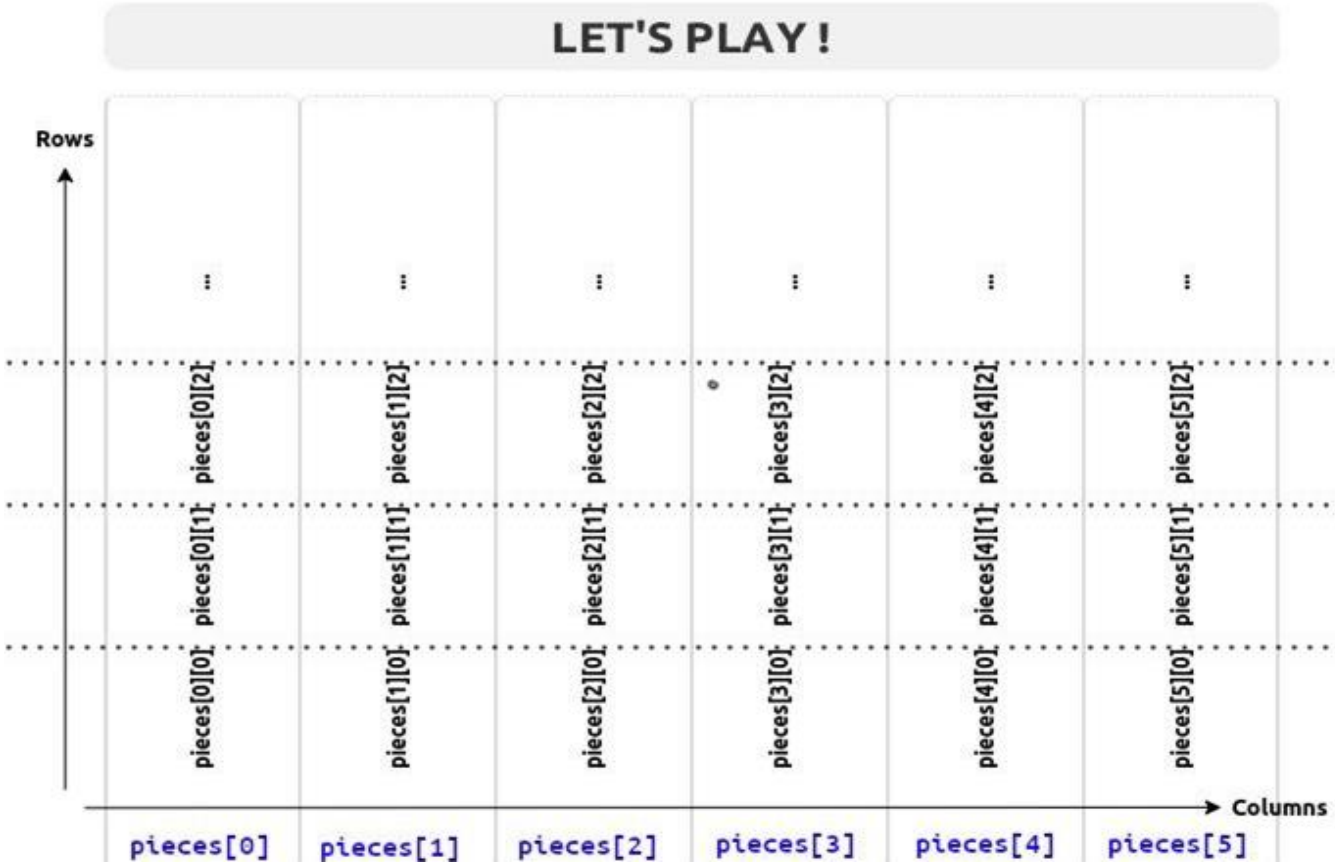
- Create **`Getters`** and **`Setters`** for each encapsulated field according to the naming conventions specified in the Java Bean Specification.
- Override the **`toString()`** method to return this,

```
return "Winner{" +
        "winningPiece=" + winningPiece +
        ", col1=" + col1 +
        ", row1=" + row1 +
        ", col2=" + col2 +
        ", row2=" + row2 +
        '}';
```

- Make sure to initialize encapsulated fields with the respective argument inside the constructors.
- Make sure to initialize `col1`, `col2`, `row1`, `row2` to `-1`, inside the constructor that only accepts the winningPiece as an argument.

`lk.ijse.dep.service.BoardImpl`

- Do not worry about implementing other methods yet, but make sure they are compilable.
- Make sure to initialize the **`boardUI`** field with the respective argument within the constructor.
- Make sure to initialize the **`pieces`** 2D array with **`NUM_OF_COLS`** x **`NUM_OF_ROWS`** (not the other way around) within the constructor.
- This is 6 x 5 Board (6 Columns and 5 Rows)



`lk.ijse.dep.service.Player`

- Make sure to initialize the **`board`** field with the respective argument within the constructor.

Once completed, you **SHOULD NOT** see any compilation errors any more

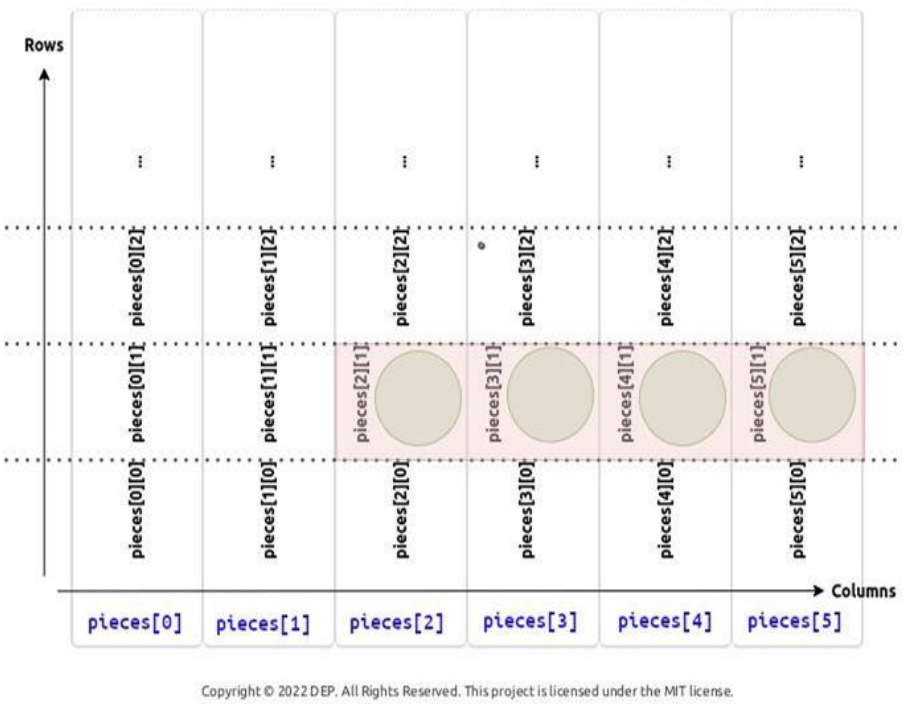
🎮 Step2: Implementing the logic (Marks: 40)

If you have come to this step, it means that you have successfully finalized the class hierarchy and overcome compilation errors. Now you may run the game if you wish, but it doesn't do anything. There are two players in this game. One player is the gamer who actually plays this game, we call him the **`HumanPlayer`** and the other one is the computer and we call it the **`AiPlayer`**. In this step, we implement **`AiPlayer`** based on randomness rather than intelligence.

`lk.ijse.dep.service.BoardImpl`

- By now you should have initialized the **`pieces`** 2D array with **`NUM_OF_COLS`** x **`NUM_OF_ROWS.`** But yet all the pieces still contain **`null`**. So we need to initialize all the pieces of the 2D array to **`Piece.EMPTY`** within the constructor.
- The **`getBoardUI()`** should return the memory location of the **`BoardUI`**
- The **`findNextAvailableSpot(int col)`** should return the first empty space in the specified column. For example, if the first three rows of a column are filled, it should return 3. Remember that the first row is labeled row 0. If there are no free spaces in the column, the function should return -1. (See the above image for clarification)
- The **`isLegalMove(int col)`** should return a boolean representing whether the current move is a legal move or not . This method can be implemented via invoking **`findNextAvailableSpot(int col)`** and then finding whether there is an open space in the row.
- The **`existLegalMoves()`** should check if there exists any legal moves on the board. Do not worry whether one side has already connected 4 in a row. As long as there is a single spot available this method should return **`true`**.
- The **`updateMove(int col, Piece move)`** method should update the board by putting the piece whose turn it is in the first free row of the specified column. You are not validating whether the move is legal here (Assuming it is a legal move)
- The **`findWinner()`** method should check if either player has already connected four pieces. It should return a **`Winner`** instance that consists of **`Piece.GREEN`**, if AI has won, **`Piece.BLUE`** if the human player has won, or **`Piece.EMPTY`** if there is no winner (in case of tied match). This method must work any time that there are four pieces connected horizontally or vertically. **Diagonal is not required to check**. When the winner is AI or the human player, the **`Winner`** instance not only contains the winning piece but also contains the coordinates (**`col1, row1, col2, row2`**) of the connected 4 pieces. As an example, according to the picture,

```
col1 = 2
row1 = 1
col2 = 5
row2 = 1
```

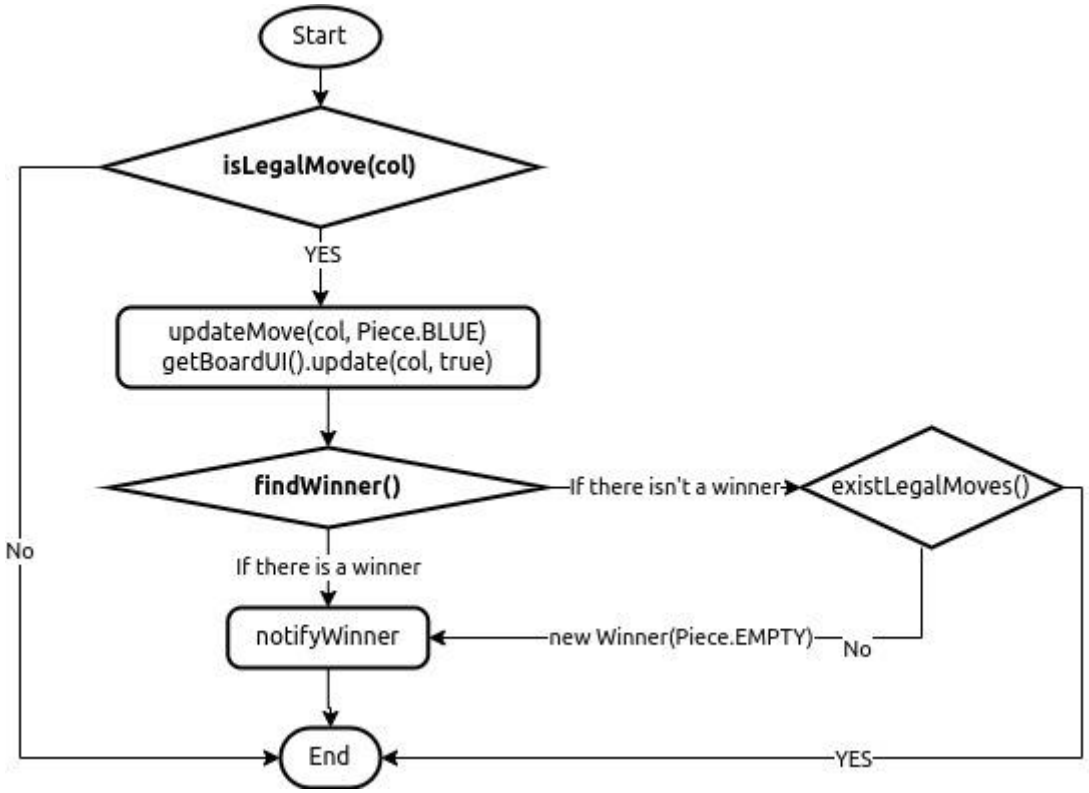


When implementing the **`findWinner()`** algorithm, it is important to prune unnecessary searching paths that may cause performance hits otherwise.

● It is not a responsible of the **`BoardImpl`** to trigger UI rendering or working with any sort of GUI stuff, it is a responsibility of the player to notify the UI

`lk.ijse.dep.service.HumanPlayer`

- When the human player clicks on a column, the **`movePiece(int col)`** method will be automatically invoked with the clicked column index, it is your responsibility to check whether it is a legal move or not. If it is a legal move, then
    - Update the board by calling board's **`updateMove(col, Piece.BLUE)`** method
    - Notify the UI about the move via **`BoardUI`**'s **`update(col, true)`** method
    - Check whether there is a winner by calling board's **`findWinner()`** method
        - If there is a winner, notify the UI about the winner via **`BoardUI`**'s **`notifyWinner(winner)`**
        - If there is no winner, checks whether any legal moves exist across the board via board's **`existLegalMoves()`**, if not notify the UI that the game is tied

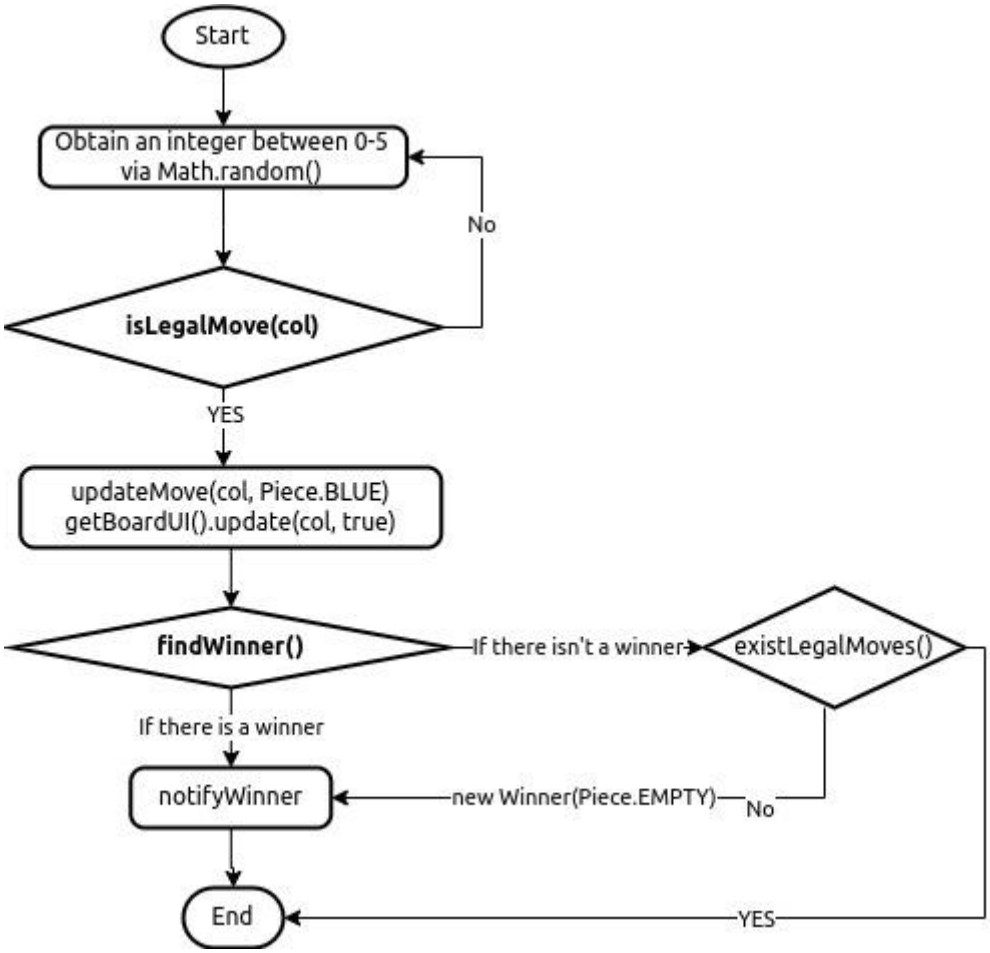


- It is player's responsibility to notify the UI as described above
- After each move of either player, the other player gets the chance automatically if there is no winner after the movement, there is nothing you have to do about that.

`lk.ijse.dep.service.AiPlayer`

- AI player's **`movePiece(int col)`** method will invoke automatically with the columnIndex **-1** once it's turn comes. There is nothing you have to do about this. It is already coded into the UI logic.
- As mentioned at the beginning of this step, we implement **`AiPlayer`** based on randomness rather than intelligence. So pick a random **integer** value between 0-5 using `Math.random()` API. You may use other utility methods in `Math` as well. **If the randomly picked column can't process a legal move, continue to find a random column that can process**

  Hint: Use `do..while` loop
- Once a legal move can be processed, Update the board by calling board's **`updateMove(col, Piece.GREEN)`** method
- Notify the UI about the move via **`BoardUI`**'s **`update(col, false)`** method
- Check whether there is a winner by calling board's **`findWinner()`** method
  - If there is a winner, notify the UI about the winner via **`BoardUI`**'s **`notifyWinner(winner)`**
  - If there is no winner, checks whether any legal moves exist across the board via board's **`existLegalMoves()`**, if not notify the UI that the game is tied



If you have come to this point which is not difficult at all (kids stuff 😉), now your game should work fine. Congratulations!

If you want to find out how the games look like after the completion of step-2, you may visit the following link and follow the instructions there,
https://github.com/Ranjith-Suranga/connect-four-game-assignment/releases/tag/v2.0.0

🎮 Step3: Implementing the artificial intelligence (Marks: 30)

Even though our game works fine now, our **`AiPlayer`** is still a dumb. Because we created our **`AiPlayer`** based on randomness not on intelligence. It is time to revamp our **`AiPlayer`** so he can act smart and make it really hard for the **`HumanPlayer`** to win the game. To do that, we are going to use a popular algorithm called "**MCTS (Monte Carlo Tree Search)**" that helps **`AiPlayer`** to make intelligence decisions.

Introduction to MCTS (Monte Carlo Tree Search)

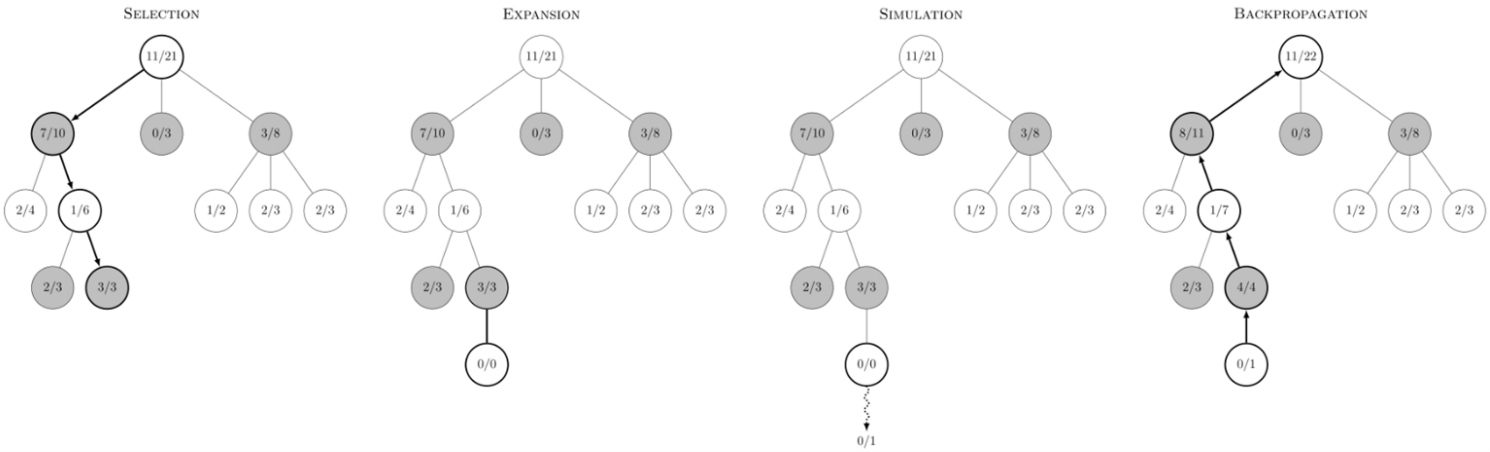
In computer science, Monte Carlo tree search (MCTS) is a heuristic search algorithm for some kinds of decision processes, most notably those employed in software that plays board games. In that context MCTS is used to solve the game tree. Each round of Monte Carlo tree search consists of four steps:

- **Selection**: Start from root R and select successive child nodes until a leaf node L is reached. The root is the current game state and a leaf is any node that has a potential child from which no simulation (playout) has yet been initiated.
  The main difficulty in selecting child nodes is maintaining some balance between the exploitation of deep variants after moves with high average win rate and the exploration of moves with few simulations. The formula for balancing exploitation and exploration in games, called **UCT1** (Upper Confidence Bound 1 applied to trees) recommends choosing in each node of the game tree the move for which the following expression has the highest value.

  $$\frac{w_i}{n_i} + c\sqrt{\frac{\ln N_i}{n_i}}$$

  wi stands for the number of wins for the node considered after the i-th move
  ni stands for the number of simulations for the node considered after the i-th move
  Ni stands for the total number of simulations after the i-th move run by the parent node of the one considered
  c is the exploration parameter—theoretically equal to √2; in practice usually chosen empirically
- **Expansion**: Unless L ends the game decisively (e.g. win/loss/draw) for either player, create one (or more) child nodes and choose node C from one of them. Child nodes are any valid moves from the game position defined by L.
- **Simulation**: Complete one random playout from node C. This step is sometimes also called playout or rollout. A playout may be as simple as choosing uniform random moves until the game is decided (for example in chess, the game is won, lost, or drawn).
- **Backpropagation**: Use the result of the playout to update information in the nodes on the path from C to R.

Here is a simple visual explanation of the above mentioned steps



Getting started with the MCTS algorithm

Most probably this may be the first time you heard about this algorithm. If that is the case, please quickly follow the resources mentioned below, it doesn't take a lot to get started with this algorithm.

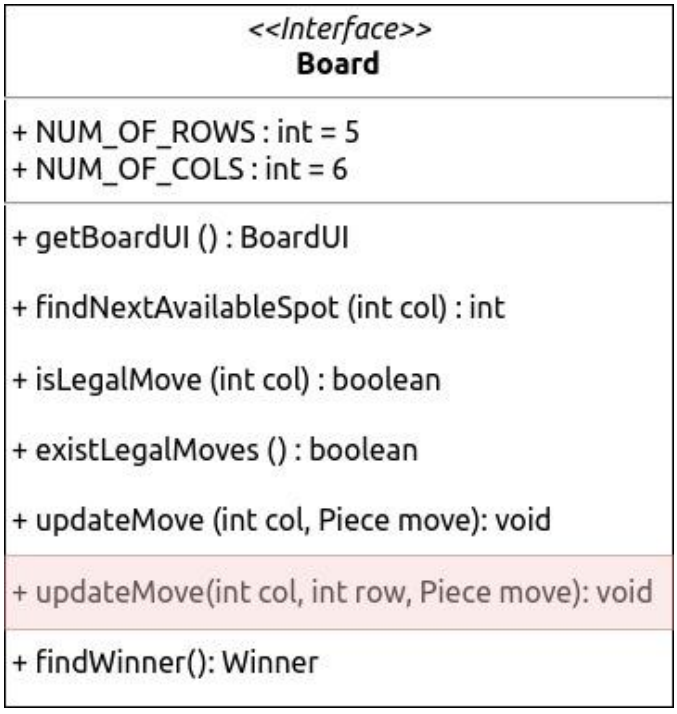
1. https://en.wikipedia.org/wiki/Monte_Carlo_tree_search
2. https://youtu.be/Fbs4lnGLS8M?si=kjwEptsHaP5LoQ0N
3. https://youtu.be/UXW2yZndl7U?si=zAWxQ9jcyEbRBVQt (Only video you ever need to watch)

Time to code

Once you understand the MCTS algorithm and how it works, it is time to code it in Java. But first, there are few changes to be made.

`lk.ijse.dep.service.Board`

- Revamp the interface as below,

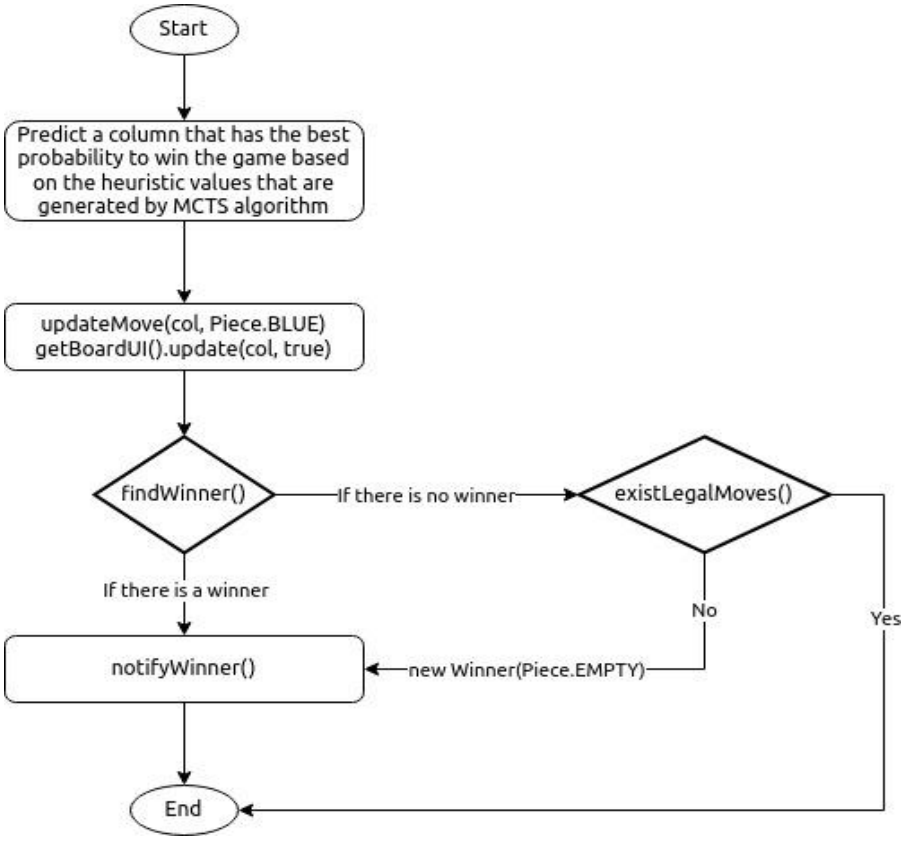


`lk.ijse.dep.service.BoardImpl`

- The **`updateMove(int col, int row, Piece move)`** method should update the board by putting the piece whose turn it is in the specified row of the specified column. You are not validating whether the move is legal here (Assuming it is a legal move)

`lk.ijse.dep.service.AiPlayer`

- You most probably need to create a few extra classes (Hint: `MctsAlgorithm`, `Node` 😶) to implement the MCTS algorithm. Ensure they all are **static nested classes** and **encapsulated** within the **`AiPlayer`** class.
- Once you have successfully implemented the MCTS algorithm, use it to obtain the **heuristic value** for each column, then based on heuristic values, predict a column that has the best probability to win the game for the AI player.
- If you want to create any utility methods to implement the above AI part, you are free to create them as much as you want within the class as long as they are **hidden** from the outside world.
- Revamp the AI player's **`movePiece(int col)`** method as below,

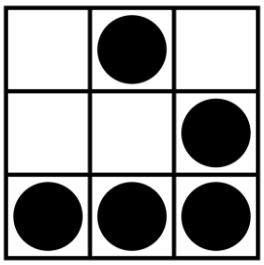


- As mentioned above, based on the heuristic value of the MCTS algorithm, the Ai player should always try to win the game, but if there is very low or no probability to win, then as a smart Ai player, it should try its best to not let the Human player win. In such a scenario, you should write your algorithm in a way that the Ai player tries to tie the game instead of winning.

🎓 Step4: Finishing off

If you have come this far, **many many congratulations**, because not only now you have a wonderful game in your hand, but also you have been graduated to the next level of the OOP world 😉

Happy hacking, see ya!



General Instructions:

1. **Pay attention to the details**
2. **You are allowed to use internet during the assignment except for the MCTS algorithm**
3. **You need to place comments on the MCTS algorithm describing the steps as much as possible, otherwise some marks will be deducted**
4. **You MUST do this assignment individually**
5. **You are only supposed to code within the `lk.ijse.dep.service` package**
6. **You are not allowed to create any top level classes other than this assignment has mentioned**
7. **In any step, if you have any doubts, make sure to clarify them before proceeding, you know how to contact me right? 😉**