specification(what the program is supposed to do)

design(how the program was planned)

implementation(how the program was written)

testing(what tests you performed)

conclusion(how satisfying the result is)

Task 1 XY-Coordinate

**Checkboundaries**

this method returns true when the current x and y are greater or equal to 0 and smaller than xSize and ySize(which in TicTacToe will always be the same as the board is quadratic). If the the above statement is not true the method returns false.

**Shift**

The method was modified in both XYCoordinate.java and Coordinate to also take a Coordinate as an argument. The method gets x and y from the coordinate and adds dx and dy respectively. Then it returns these new x and y values as a XYCoordinate

Task 2 TTT-Board

**isFree**

method is passed a Coordinate and get x and y values from it. If the board at those values is = 0 it returns true, otherwise it returns false. As the game uses an int array to represent the board a 0 value means no player has made a move on that tile and any other number means that player has made a move on that tile.

**getPlayer**

method is passed a Coordinate and returns the int found on the board at that position.

**addMove**

receives a Coordinate and an int representing the current player. The method assigns the players value to the board at the x and y value that it was given. This method is coded with a try catch block. If an illegalArgumentException occurs the catch block divides it into one of three categories. First block uses checkBoundaries to see if the argument that was incorrect was the Coordinate, second block checks whether the player number is valid based on the size of the board(as the board is constructed by setting the arraysize equal to amount of players +1 I.E. 2 player game results in a 3x3 board). Last block is in case something unexpected happens. Always fail loudly so the bug is easier to find.

**checkFull**

starts out with a Boolean variable set equal to true, then using 2 for loops runs through each position on the board and checks if the value at each position is equal to 0. If it is the Boolean is set equal to false, and at the end of the method the variable is returned. In this case if the method just once encounters a board position that is equal to 0 the method returns false.

Checkfull is used in checkResult and here the if statement is changed to an else if statement. Before in cases where a player won on the last possible move the game would show the result screen with “this was a draw” printed.

**checkwinning**

the method uses 3 different double for loops to iterate through all possible starting positions used for checkSequence. First block checks rows and columns, here the important bit is that for columns is that we don’t need start positions in the 2 right most columns, as there aren’t enough tiles for a full sequence of 3 markers. The same goes for rows but here we don’t need start positions in the bottom 2 rows.

The other 2 double for loops create start positions for checking the diagonals. These are divided into 2 groups, one for diagonals going up and one for diagonals going down. Starting positions for the first group is a combination of the ones for rows and columns. We don’t need positions in neither the bottom 2 rows or the rightmost 2 columns. For the second group we don’t need the top 2 rows and the rightmost 2 columns

The method then calls checksequence with the generated start positions and dx and dy corresponding to either rows(dx=1 ,dy=0), columns(dx= ,dy=1) etc.

if the returned value from checksequence isn’t 0 it means that a player has won and checkWinning returns the int.

**checksequence**

takes 3 arguments, a Coordinate and 2 integers, dx and dy. The method creates an int variable, called checkValue, equal to the board at the Coordinate and if it’s equal to 0, returns it. Then using a for loop to run the shift method twice with the given dx and dy arguments. After each loop it checks whether the board at this new position is equal to checkValue. If the are not equal the method returns 0 and if they are equal the second loop occurs and a new check happens. Again if they aren’t equal o is returned, but if they are equal it means 3 tiles in a sequence are the same and that player has won, resulting in checkValue being returned.

Task 3 testing the Game –

**Checksequence method:**

an array out of bounds error occured in the checksequence method, this was caused by updating the start coordinate. so first run through x,y was 0,0 then 0,1 then 0,3 and should have been 0,2 in the last, but start pos was 0,1 and i shifted with 0,2 = 0,3. was discovered using log.d. fixed by deleting the start update, but could also be fixed by only shifting by 0,1 both times.

Later changed to only shifting by 1 each time and updating the starting pos.

**Error in the checkwinning method:**

Currently you can only win the 2 player game by placing a marker on one of the corners. Different win conditions on 3+ players, but still cant win by having 3 in a row/column/diagonal.

Part of this error was in the checksequence method. But now the win condition only happens with diagonals in 2players and in 3player diagonal wins only happen in the far corner. Only 1,2 2,1 3,0 and 1,1 2,2 3,3 results in wins.

Issue was that I was checking the winner variable outside of the for loops so only the last winner result was ever checked. Fixed by placing the if block inside the loop.

**CheckResult method:**

There was 2 if statements, one for checking for a winner and one for checking if the board was full. This resulted in the case that where a player won on the turn that the board was also full the message would state that the game was a draw(I assume it for a split second actually said “player x wins”, but that it was overwritten with the draw message). Changing the full board statement to an else if means it only gets run if there is no winner.

AI.java:

First issue was trying to get a board passed from TTTGame to use for evaluating the best move for the ai, but board in TTTGame isn’t a int[][] so fuck that shite. Instead had the addMove method in TTTBoard call the aiAddMove method and pass the current board along.

Value bug: having a lossValue of 80 resulted in a few cases where a winning move and a blocking move had the same value(winning move = 100 – 10 for a oppmarker, and blocking move being 80+10 for own marker).

Fixed by reducing lossValue to 79

Human start 1,1: if the human player starts and places a token in 1,1 I.E. the middle. All other free positions where evaluated to -10 and as bestValue starts of at 0, none of the free positions returned a higher value than 0 a bestMove was never set. **Result= nullpointer on getX() in addMove();**

Fix:

**ConnectFour:**

Bug: nomatter where I clicked player 1 won. No mistakes found in checkwinning and checksequence. The bug was in how the marker was placed. For loop ran through Y-values from high to low(bottom to top) and the first time an empty cell was encountered placed a marker there. It was missing a “break;” statement so it ran through all y values even after placing the first marker. This lead to having 4 in a row after the first click and player 1 winning. Missing break found by always return 0 in checkwinning and seeing the entire column filling with the current players number instead of only the lowest tile.

Task 4 AI

The AI was planned to analyze the board and give each position values based on how good likely it was that it would lead to a win or at least a draw. It was to do this by iterating through the board and for each position look at the row, column and diagonals, where applicable, and add values together and then choose the position that had the highest value. At first the idea was that recursion should be used to get the AI to look several moves ahead to but initial testing revealed that already while just looking at the next move there was only a few situations where it didn’t guarantee at least a draw.

It was only written to work for a 2 player 3x3 board but could be extended to work on both larger boards and more players, of course here we might not be able to guarantee at least a draw.

The Values used to analyze the board were to start with 100 for a tile that would lead to a win, 90 for a tile that would lead to a loss, and 10 and -10 for tiles containing the AI’s mark and the players mark respectively. Later the lossValue was changed to 79 to avoid edge cases where the AI could win by placing a mark in one spot but choose to instead block the player in another spot.

The important methods in the AI are generateMove, moveEvaluate and the 4 check methods.

The 4 checkmethods check each row, column and if applicable one or both diagonals and returns a value based on what each tile in that sequence contained, I.E. if it had 2 AI marks it would return 100.

MoveEvaluate simply adds the 4 check methods together and returns it.

GenerateMove runs, using 2 for loops, through all 9 possible positions on the board, and if that position is free(I.E. contains a 0) calls moveEvaluate. The integer returned here is assigned to the variable currentValue, which is then checked vs another variable, bestValue. If both have the same value a new XYCoordinate is created using the current x and y values in the loop. This Coordinate is then added to bestMoveList. If currentValue is greater than bestValue bestMoveList is cleared and then a new Coordinate is added. This is done so that if several possible moves have the same value all are saved in the list and a random one is picked in the end.

There are three exceptions to the standard rules that the AI follow.

The first is that when the AI goes first it will always place a marker in the middle, as this is where there most possible paths that lead to a win for the AI.

The second is if the human player starts and places a marker in the middle, then all other 8 positions will return a value of -10 and therefore currentValue will never be greater than or equal to bestValue. This results in no move being added to the bestMoveList and a NullPointerException. The solution we initially chose was to set bestValue to -10 and call the generateMove method again, but this resulted in a possible loss if the AI chose to place it’s marker in one of the middle tiles. Instead we not just populate the bestMoveList with Coordinates for the 4 corners.

The last is an extension of the second exception. In the case where the human player made a move in 1,1, the AI would the mark a corner, and if the player then marked the opposite corner the result would be that the AI would force a loss on itself with the standard values (unless the player made a mistake). The solution was to implement a check that would look for this case and then alter bestMoveList to contain 2 different Coordinates.

***TODO picture here!***

Task 5 ConnectFour