UNIVERSITÄT DUISBURG ESSEN

NES Networked Embedded Systems

Programming in C/C++

Exercise Sheet 08
!!! Deadline: Tue, Jan 7th, 23:55 !!!

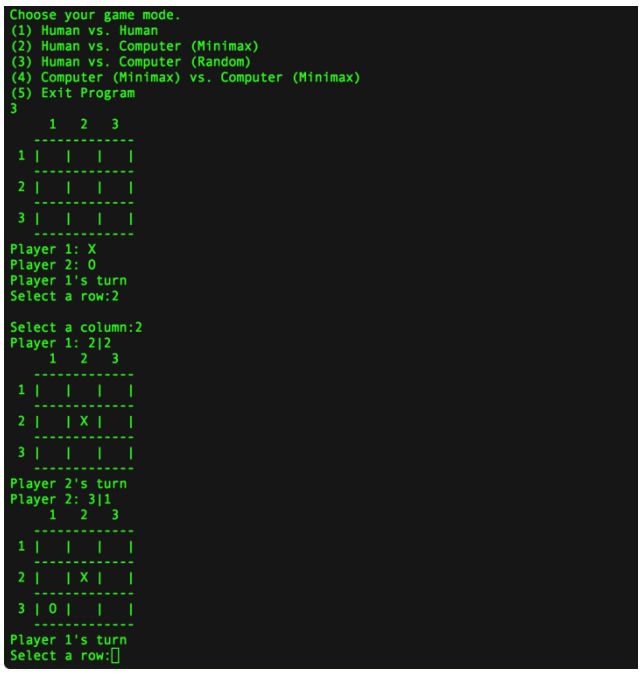
Task 01:

In the following tasks you will write a TicTacToe game with an AI. It has the following requirements:

i. The game has two Modes:
   a. Human vs Human
   b. Human vs Random AI (which makes random moves)
ii. Uses a regular 3x3 field which is printed to stdout after each turn. **Use ASCII art.**
iii. Player 1 uses 'X' and Player 2 to uses 'O'
iv. You **have to** use C++ inheritance for the different player types. Don't use duplicate code.
v. After the game is over you should be able to start a new game or to exit the app.

Your output should look like displayed in the picture. Use the **exact same menu structure** as displayed, even though minimax is not implemented yet. That will be done in the next task.

When a player enters a wrong column/row combination (either because it's outside of the field, or because the spot is already used by another player), print out a message like "`Invalid field selection`" and ask the player again for a row/column combination.



The UML diagram at the end is provided as an aid for you in structuring your game. It is a generic UML diagram which you should interpret for the C++ language. You can extend it to add whatever you need to implement your game. You are also not required to use this structure, feel free to implement the game as your prefer. However, the interaction with your game **MUST** be as defined above.

When the games ends, print the winner to a new line. For example:
`Winner is: Player 1 (X)`

If neither Player1 nor Player 2 wins, print:

```
A draw!
```

Test your game. Set “-std=c++11” in the Makefile, if needed. Play against the AI and other students. Remember to comment all your methods and classes properly.

Submission: Your executable should be called **tictactoe**.
Points: (code 60pts, comments 15pts)

Task 02:

In this task you will extend task1 with a new mode, which is an AI Computer player using the Minimax algorithm.

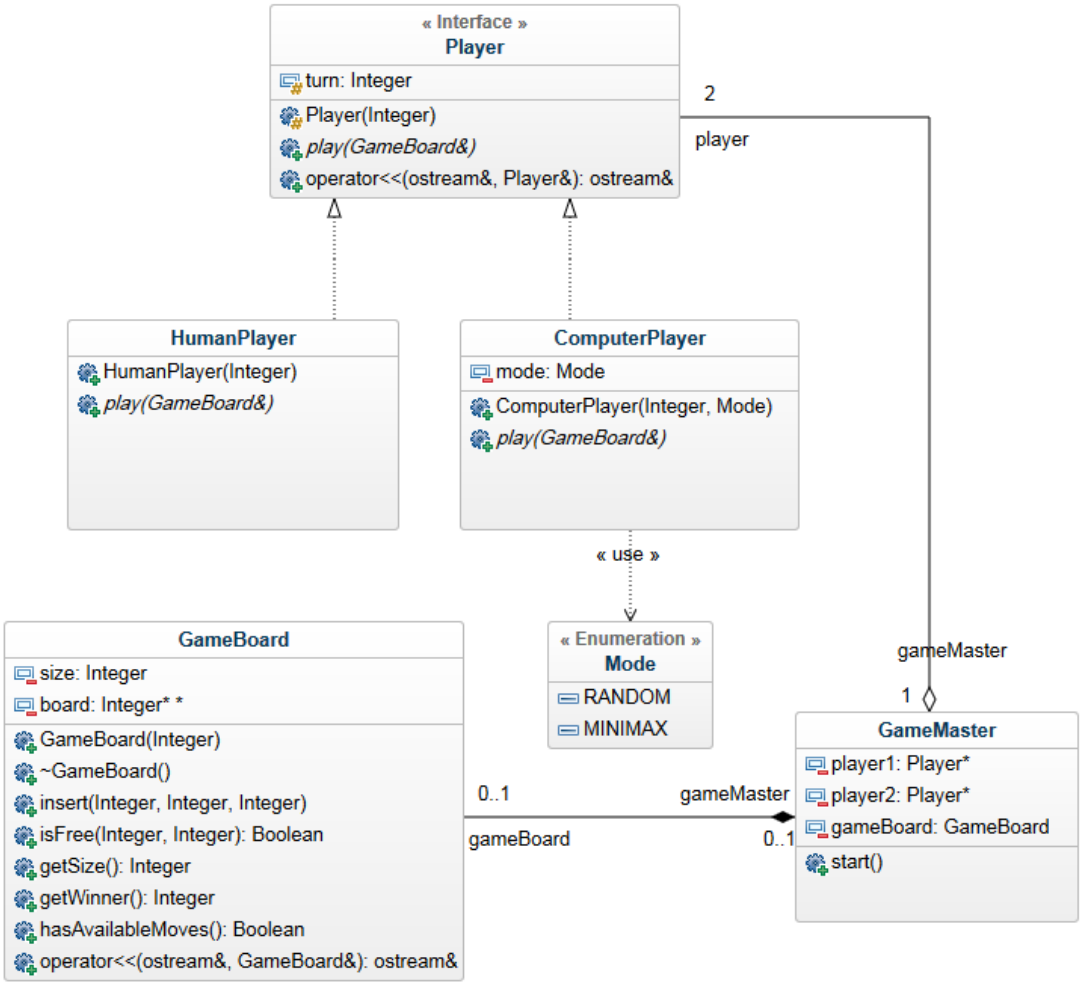
The minimax algorithm can be used to determine which move is the best for one player and the worst for the opponent. Minimax is also known as full tree search.

You basically check all possible moves from both players to find out which one is the best for you or the worst for the opponent. So the AI always makes a perfect move.

Test your game. Play against the AI. If you implemented the minimax algorithm well, you *may not be able to win*. A good player against a well written minimax AI will almost always end in a tie.

Comment all your methods with doxygen-style comments.

Submission: Submit another copy of your code as a new task with its own Makefile.
The difference to task 01 submission should only be the minimax algorithm.
Your executable should be called **minimax**.
Points: (code 25pts)


« Interface »
Player
turn: Integer
Player(Integer)
play(GameBoard&)
operator<<(ostream&, Player&): ostream&
2
player
HumanPlayer
HumanPlayer(Integer)
play(GameBoard&)
ComputerPlayer
mode: Mode
ComputerPlayer(Integer, Mode)
play(GameBoard&)
« use »
« Enumeration »
Mode
RANDOM
MINIMAX
gameMaster
1
GameBoard
size: Integer
board: Integer* *
GameBoard(Integer)
~GameBoard()
insert(Integer, Integer, Integer)
isFree(Integer, Integer): Boolean
getSize(): Integer
getWinner(): Integer
hasAvailableMoves(): Boolean
operator<<(ostream&, GameBoard&): ostream&
0..1
gameMaster
gameBoard
0..1
GameMaster
player1: Player*
player2: Player*
gameBoard: GameBoard
start()