ConnectN

To make the game ConnectN, significant structural changes need to be made. Large parts of the code already make inroads towards ConnectN, but further changes are still needed.

This new revamped code will be able to:

1. Play ConnectN (2<N<m, for integer m less than both board side lengths)
2. Played by any number of Human and computer players.
3. To be played on any game board size (within reason)

This new program will have many classes:

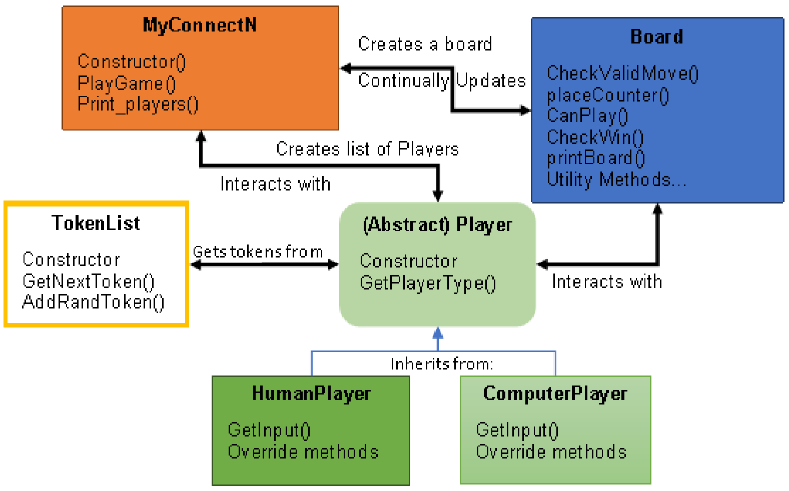
- **Main** – The main, to create the MyConnectN class.
- **MyConnectN** – Main game program, interacts with all the others to run the game. Creates a **board** and a list of **players** and loops through them to perform players' "turns".
- **Board** – To store the current game happenings. Includes methods to see who wins, the state of the board, where you can play, etc.
- **(Abstract) Superclass Player** – A player that never plays, creates and stores some useful methods for its child classes.
- **HumanPlayer** child of **Player** – A Human player, allowing input for when the player needs to make a move
- **ComputerPlayer** child of **Player** – A "Bot", Making moves at random
- **TokenList** – A list of tokens that the **players** will choose from, when it runs out of tokens, will get a random lowercase letter that isn't already in play for a **player** to use.

We take advantage of modularisation and other fundamental OO concepts to make this game possible. We further modularise the code to split up routines that no longer need to be together.

We can also reduce code re-use by making some of the methods very generic, able to work with any player's token and win condition.

A few more properties are also added to the **MyConnectN** class, so we have:

- **Board** board
- List<**Player**> players
- **Token_list** token_list
- Num_comp_players
- Num_human_players
- N_in_a_row



The field "win_n_in_a_row_condition"; allows us to alter the check_win() method to work with any N, thus needing only 4 different ways to check the victory conditions. We also can make the method more generic to check for the players token (not looping through each player). So, we don't need 2 methods per win condition.

We can also introduce a method check_valid_move() to check if the players input is a valid move, both checking its within the board dimensions, and if they can place it there. A method to check the counters in a column is also needed here (if a column is full; we can't place a token). We also want to check the players type in check_valid_move(); as if they're a computer player, we don't want to print an error message warning them. We also need to alter the print board method to print more than just 'r' and 'y' tokens (as we can have any number of players).

From before, we still have "player" as an abstract class while the Human/Computer Players are still children of it. This allows us to reuse important code for a class which can never be initialised.

To allow for any number of players, we need to alter the way that MyConnectN runs the game. We can make a list of “players” which can be populated by both human and computer players.

The List of players can then inherit the useful methods from the list class. The MyConnectN class then loops through each of the players in turn, allowing them a move on the board and checking to see if they’ve won. The pseudocode aside describes how this could work practically.

```
Playing Connect N

Define Board as a new Board (using input values)
For int in number human players:
    {Add Human player to list, give token from token list}
For int in number Comp players:
    {Add Comp Player to list, give token from token list}

While (no one has won/it's not a draw):
    For each player in playerlist:
        Check they can make a move, if not: It's a draw
        Get the Players Input for a move
            if it's a valid move: continue,
            if it's not a valid move: make the player input again
        Place the players token on the board
        Check to see if the current player has won
        (checking for N in a row, diagonals, horizontals and verticals)

        if the current player has won:
            End the game, print that the current player has won
        If not:
            Move onto the next player,
            if current player is the last player in list:
                go back to the starting player
```

To allow any number of players, we also introduce the “TokenList” class. This class stores all possible tokens and creates new tokens for a player (a token that currently isn’t in play) if necessary. This allows us the flexibility of up to 26 players.

We also add 2 loops at the top of the playGame() method, adding the human and computer players to the players List, and handing them a token from the token_list class.

Overall, all these changes should make the code concise and succinct, reducing repetition and making it easy to test and change. The interaction of classes is kept at a minimum to keep coupling low.

A large improvement of the code is making it more object orientated, following the principals of the course we can also make the board/game/player classes more generic. We then can have a more robust and changeable program. Allowing inheritance and polymorphism allows us to use the Human/Computer player classes to use the superclass’s methods and properties, and we see that these objects take multiple forms. Override methods can then be used to tailor the child class to our needs.

To make the game class more flexible, we can add another constructor method with more input parameters. This means we can initialise the game from the Main() method with customised inputs, such as the number of computer and human players, the board size and the win condition.

Further Extensions

Further, we could alter the game to:

- Allow different computer behaviour, incorporating simple “strategies” for the bots, instead of just placing a token randomly on the board. We could do this by making more child classes from the ComputerPlayer, keeping the basic methods but overriding the input method. E.G:
    - Placing a counter -1/0/+1 from the last placed counter; to allow focus on a single part of the board.
    - “Mirroring” the Human Player, this strategy works for some board sizes.
    - ConnectN is also a “solved” game, meaning there is an optimal strategy for each board size and each state of game. You could insert an AI which would always win.

Summary

The code I have submitted with this report matches these requirements and specification. It also matches the basic Connect4, you can specify what type of game you want to play with changes to the parameters in Main().