The Myconnect4 program could be restructured in many ways. I will aim to restructure the code mainly using modularisation/encapsulation and inheritance. This will create many classes instead of running the game in one function. These classes with interact with each other and some will be child and parent classes. I aim to provide a low amount of coupling in this implementation.

Modularisation:

Modularisation subdivides much of the programs code into separate subroutines or programs. This allows you to write more robust code as you can test each module separately from the main program.

I believe the following methods in the MyConnectFour program could be modularised:

- Checking the win condition for vertical, horizontal and diagonal
- Separating out the players actions from “playgame”
- Checking if a player’s input is a valid “move”
- Printing the board
- Checking the number of tokens in a column
- Checking if the game can be played at all – or if not; then it’s a draw
- Getting the Length and Width of the board
- Performing a “move” of a player
- Getting a player’s “token”
- Getting a player’s “type” (if computer or a human)

Creating subroutines also allows us to write more generic and reusable code. Reducing code length and making it portable to other purposes. This then gives us the opportunity to further adapt the code later to do things such as: Adding more players or changing the win condition N.

Encapsulation:

Encapsulation wraps data and methods together in a single place. We can also hide the fields of the class from other classes. These private fields can only be accessed from the class’s methods, leading to ‘safer’ code. Using private fields and public getter/setter methods allow a programmer to achieve better encapsulation. This is a form of abstraction which allows us to hide all the inner workings of the game from the players, they instead interact with the game via the command line, which is used as input for the relevant methods.

The code currently is not very encapsulated, work is needed to separate out the parts of the program into separate units.

- Make the “board” its own class with a printing method, getting its size, etc.
    - Extract the print_board method
    - Extract the placing of a token (with class “player” as an input)
    - Make a method to check if a player can place a token using the number they input
    - Checking the win conditions
    - Getting the length and width of the board
- Make each player their own class and how they interact with the board.
    - A method to get the users input
    - A subroutine to get the players token
    - If they’re a computer player, we can get a valid move from the board and pass it back to the game
- The MyConnectFour class, we can make this class more customisable, these can be done with the right fields and changing methods mainly in the board class.
    - Adding a field for the list of *players*, we can loop round the players performing each of their turns
    - Adding a field for the *board,* this will hold the current state of the game and allow us to play the game. We will initialise a new board in the constructor for MyConnect4

Inheritance

This allows a class to acquire all the methods and properties of another class, helping reduce repeated code. This also introduces polymorphism where the “player” subclasses are also members of the “player” class. We can then take advantage of all the super-classes methods and properties, and how other classes interact with them. We do this by:

- Having a superclass “player”. Using Hierarchical inheritance to allow 2 subclasses to extend the superclass “player.
    - ComputerPlayer and HumanPlayer will be subclasses of the class “player”
    - They inherit most of the basic methods, like getToken, placeToken etc.
    - But the computer player will have a simple move set with what it can do.
    - The Human player, in contrast will get input from the user for its moves.

Inheritance should be used in the program. We have common classes where we can take advantage of the same methods. This greatly reduces repeated code and allows us to benefit from all the hard work we’ve done previously.

Abstract Classes

Abstract classes are classes which cannot be initialised in the program. However, they can provide abstract methods (methods without any code), for child classes. These are thus a “base” for subclasses, meaning we can use the parent’s methods and properties, while overriding some methods if needed. We could implement abstract classes here by:

- Making the “player” class abstract, as we will never have a player which isn’t a human or a computer.
    - This means we can extend this abstract “player” class, but we don’t have to write methods which will never get called.
    - Thus, we can write GetToken, GetNumberMoves in the parent abstract class.
    - But we then have abstract methods:
        - GetInput (for the Human/Computer player’s move)
        - GetPlayerType (to know which type of player this is)

I think we could use abstract classes here as we want to share code among several closely related classes. We also expect that the player subclasses that extend our abstract “player” class have many common subroutines.

Interfaces

The advantages of interfaces are that you can extend multiple interfaces for the same class, whereas you can only ever inherit a single class. For this reason, I don’t think interfaces will be used in this project, as we don’t need a class to extend multiple classes. The ComputerPlayer and HumanPlayer will inherit the “Player” class, and we don’t need any other methods from elsewhere. We also don’t expect that unrelated classes want to use a single interface.

Another use is for “tagging” interfaces. This is to either create a common parent or to add a datatype to a class. Again, I don’t think we need either of these purposes in this project.

Extensions

To extend the game, we could:

- Change many of the methods to work with: *n_in_a_row* a new parameter that reflects how many tokens you need to place in a row for victory
- Allow any number of players, not just 2.

Summary

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