Coursework 2 – Principles of Programming 1

Modularisation

Modularisation refers to the breaking of large chunks of code into smaller, more manageable pieces of code. The modularised code would allow the user to edit and test the methods independently, without affecting the rest of the program. In OOP, modularisation could be implemented in the function scope as well as the class scope.

The original code was not a good example of modularisation, since several concepts could have been extracted and put into their respective functions and classes. Examples include –

1. No separate method for checking the vertical and horizontal winning conditions.
2. No separate method to check for if the player has won the game.
3. No separate classes for the board and players as they represent complete standalone entities in the code.

In the improved code, the players and boards were extracted into standalone classes to represent the respective entities as objects. Similarly, in the classes, similar concepts are extracted into their own methods, for e.g., in order to check if the player has won the game, the horizontal, vertical and diagonal checks in the GameLogic class are modularised into their own class and called from a single method, the checkWin() method. The checkWin() is the only method that is accessible to the GameController class (an example of encapsulation), however, having each win condition in their own class makes it easy to maintain and test the particular functionality.

Encapsulation

Encapsulation refers to bundling of related data to organise the code and maintain a degree of separation which would allow one to change the code without affecting the other parts of the program. On an abstract level encapsulation refers to the hiding the inner functioning of the object by just providing an interface for the user to interact with.

The original code followed encapsulation to a certain extent by having private access modifiers and methods in order to hide data within the Connect4 class (such as char[][]board property).

However, the principle of encapsulation could be better implemented by a GameController class to encapsulate the flow of the game, allowing the Main class to only access the startGame() method. Similarly, the Board and the Logic of the game could be encapsulated in their own classes and accessed only by the GameController class. Another example of the use of encapsulation in the modified report was to implement getters and setters for every method, which would not allow the classes to access the properties of another class and only access the getter and setter methods. Getters and setters not only hide the implementation of the properties, but also allows for easy reuse.

Polymorphism

Polymorphism allows an object to display multiple forms of behaviour. For example, an example for polymorphism would be the representation of various nodes such as Node2D or Area2D as a GameObject in a game engine such as Unity. In Java, polymorphism can be implemented by using inheritance, abstract classes and interfaces. The choice to use any one of them depends on the specific requirements of the different players.

The original code is not a good example of polymorphism, especially for the required specifications, since there is no provision or architecture in place for the player to be a computer, or the other entities to be anything other than their defined and expected behaviour. The game could have multiple types of players such as computers and humans or a combination of them. Another implementation of polymorphism could be creating an abstract Board class and have a the standard 6x7 Connect4 Board inherit from the Board class.

The improved code applies polymorphism by implementing a Player interface, implemented by the HumanPlayer and ComputerPlayer class. The usage of the Player interface would allow the GameController to treat all kinds of classes which implement the Player interface in the same way. An example of this could be seen in the property List<Player> players, which keeps a list of the players only if they implement the Player interface. Another example could be seen in the "switchPlayer (Player currentPlayer)" method which switches the player; however, this would only work if the both the players shared a common interface.

The choice for creating an 'Player' interface, instead of an using an abstract class or inheritance was dictated by the lack of methods common for the 'HumanPlayer' and 'ComputerPlayer'. Examples of methods which had to be implemented in 'HumanPlayer' and 'ComputerPlayer' included makeMove(), getters for player name and mark, and constant fields. These methods might be common for both the classes, however, other than the getters and setters, the functions have different implementation for different kinds of players.

In the current version of the game, the 'ComputerPlayer' object only places the counter randomly, however, for future scalability, the user might want to add computer intelligence, and have different computer classes such as 'ComputerEasy' and 'ComputerHard' which implement from the 'Computer' as well as 'Player' interface. The limitation of having a single Abstract class for an object would not allow multiple objects to have different classes from which they can inherit from.