Principles of Programming: Coursework 2 – Requirement 2

Connect 4: Report discussing code restructuring to reflect fundamental object-oriented program concepts.

Submitted by: Shayaan Khatri

Username: ssk57

To successfully implement the Connect4, keeping in mind the fundamentals of object-oriented programming, firstly I will debug the program as per the error list so that I have a working version of the game Connect4 to work with.

After I have a basic working version of the game functioning, I will focus attention on the code design and code structure. At this point I would have my entire code in a large class file that contains all the method. This is against the fundamentals of OO programming as such as code is complex, not flexible and the benefits of OO programming are not being utilised. To make my code more clean, maintainable and flexible, I would divide the current code into parts and each part can be a different class. Then I would use a Main class to bring all these classes together. This is the concept of **Modularisation.** Each class can be considered a single deployable unit. The code would then be independent and reusable due to this. Partitioning my software in different classes would help me reduce system complexity, optimize code development and minimize coupling. The Connect4 game could be divided into several parts such as Board.java, Token.java, Player.java, etc. However, the key to achieving modularity is through Encapsulation. Encapsulation is a fundamental concept of Object-Oriented Programming. It describes the idea of bundling data and methods that work on data within a class in Java. This is used for "information hiding", to hide the internal state from the outside. When encapsulation is used in a class, the variables are declared as private, however a getter method is used to retrieve an attribute and a setter method is used to modify this attribute. Keeping these concepts in mind, if I want to hide a variable in my class, I will declare it as private and code the getter and setter methods wherever needed. For example, in my Token class, a char token can be declared that is set to private, however a method such as getToken() set to public can be implemented to retrieve its value outside the Class. So in my Board class I could use this getToken() method whist writing the print board method, so the empty tiles on the board can be replaced with tokens of the player such as "| R | " replacing "| |". I will have access to this method as the way I have decided to structure my code these empty tiles will be empty tokens. In the Board Class a variable board would be declared which is an object of the token class. Another example would be that, if I have a Player Class and I declare a variable called "playerInput" that is set to private. I would then define a getter method for this variable such as "getUserInput()" so that this input can be obtained on another class by using this method. So, by applying encapsulation, I will be able to perform modularisation such that my software's components may be separated and then recombined. I would also apply the concept of method overloading and constructor overloading whenever needed. This restructuring of the code would reflect the fundamental object-oriented program concepts.

Inheritance in Java is a mechanism in which a Child class (also called a Sub Class) can inherit all the behaviours and property of the Parent class (also called the Super Class.) This is a 'is-a relationship' which is also known as a parent-child relationship. The subclasses that are built upon an existing superclass can inherit methods and fields of the superclass, whilst the subclass can also add their own methods and fields. This provides reusability. I could apply this in my Connect4 program. I could have a 'Player' Class that is a Super Class and the HumanPlayer Class and ComputerPlayer Class can be Sub Classes of the Player Class. This would be the case of Multiple Inheritance. I could implement this by using the extend word in the subclasses such as "Public Class HumanPlayer extends Player { " and similar for the ComputerPlayer subclass. This would allow the subclasses to inherit relevant methods from the superclass, however these subclasses would also be able to have features relevant to them. For example, whilst having all the methods and fields of the Player Class, a HumanPlayer class would also have a method to ask the user for input while the ComputerPlayer class would have a method to generate CPU input that is based on an algorithm that is assigned to it. If the Super Class which is Player Class was not being used, I would have to write the similar methods again and again in both the HumanPlayer Class and ComputerPlayer Class and that is not efficient and against the fundamentals of object-oriented programming.

Data Abstraction is the process of only showing essential information to the user and hide certain details. An Abstract Class in Java is Class in Java that cannot be used to create objects and it must be inherited or accessed

from another class. In my case and building upon my previous example of the Player Class, I could further improve my code by making the Super Class “Player Class” abstract so when my code is being re-used the user of my code does not accidentally create an object from this Super Class. The Player Class will only be used to define and enforce a protocol. Building on this further, an Abstract Method is Java can only be used in an abstract class, and it does not need to have a body. The body is then provided by the subclass that has inherited the abstract parent class. In our example, since both my subclasses HumanPlayer and ComputerPlayer require an input however, the procedure to generate this input is different in both the classes. So keeping this in mind, I could have an abstract method to get user input like “public abstract getUserInput()” in the Abstract Player Class, however the body of this method will provided in the subclasses.

Interfaces in Java are used to achieve abstraction in Java, and an Interface is a collection of abstract methods. Interfaces only contain definitions and methods of a Java interface that are implicitly abstract and cannot have implementations. Interfaces are mainly used for future enhancement to the program as opposed to abstract classes which are used to avoid independence. In my case, I feel that the abstract Classes are providing me with the optimum abstraction that my program needs as discussed before. Interfaces are slower and more limited that abstract Classes and moreover, interfaces should be used multiple number of times otherwise there is no use of them. In my code structure I am only using abstraction in the case of Player Class with subclasses HumanPlayer and ComputerPlayer, so the use of Interface would not be optimal for my code. So, I will not be using an Interface in my implementation of Connect4.

While restructuring my code, I will be considering all the concepts that have been discussed above to get a working implementation of the game Connect4 that reflects fundamental object-oriented programming concepts and techniques.