There are several ways to refactor the connect4 sample code to make it easier to scale, debug and maintain. Importantly, applying these fundamental concepts of object orientated principles from the onset of the product development allows the project to be agile enough from the onset to build new functionality iteratively and efficiently. This report will outline the key recommendations and then prove their effectiveness with a functional desktop GUI based game.

**Modularisation**

The current connect4 codebase is a single class of logically separate methods. The connect4 game should implement separate modules of concerns that focus on singular functionality. The game can be broken down even at its most basic functionality into separate classes as follows:

|  |  |
| --- | --- |
| Module | Scope |
| Game Board | Size, shape, limits |
| Player creation | functionality, representation |
| AI behaviour | enablement, actions |
| User input | processing, validation |
| Board update | Visualisation |
| Win condition | Rule implementation |
| Game Controller | Class connector |

We have taken this further in our implementation. In doing so, classes are developed and tested independently. This makes debugging and updating simpler. We created, for example, unique GUI classes to handle the welcome screen, game screen and end screen. They are able to inherit from a parent GUI so as not to repeat code but remain functionally distinct so can be deployed into the code iteratively. This allowed for a modular development cycle, working as building blocks as the program features are enhanced. Finally, modularisation is a requirement for other powerful object orientated principals to be used.

|  |  |
| --- | --- |
| Class | Class Type |
| Main | Default |
| Board | Default |
| GameController | Default |
| GUI | Abstract |
| WelcomeScreen | Extends GUI |
| GameScreen | Extends GUI |
| WinnerScreen | Extends GUI |
| ColumnFull | Extends GUI |
| BoardFull | Nested Local of Column |
| WinCondition | Nested Local of GameController |
| PlayerInterface | Interface |
| Player | Default |
| AI | Nested local of Player |

**Encapsulation**

The current code base has functionally no encapsulation which can lead to several control gaps in regards to data integrity. Although there are a couple of private variables, with the code being ultimately a single object of a single class, it is private to no one. Encapsulation is to implement data protection and control. In using this paradigm, we restrict how we can change and access methods/fields of a class.

Consider the protected ArrayList<Color> variable ‘colours’ in the GUI class. This represents the available player colours that our GUI classes are going to implement. We need to agree a means of communication and updating in a common language between our classes. Code relies upon stability; we build based on expected behaviours and data types. When we originally created the abstract class GUI ‘colours’ was a Color[] class type. This variable was protected, so accessible directly by GUI subclasses, but required accessors for other classes. The Player class would use an accessor method to return a value from the colour array so that it could assign an agreed supported colour to a player object. This created consistency between the Player class responsible for organising and creating players, and the GUI for updating visually based on their colour. Color[] is limiting as the array size is fixed. We initialised it with only two elements and realised this was limiting our future development potential. We refactored our code and converted to an ArrayList class with the parameter Color. This is a huge change for all the classes that were relying on the Color[] array (Player & GameController mainly). We would have to, in theory, refactor all the methods in those classes to take an ArrayList class as a parameter rather than an Array. Instead, we have the accessor in the GUI through encapsulation. We simply modified it to return a Color[] class type by converting the ArrayList. We also made a new additional accessor to return the ArrayList. This is a simile for when third party tools get updated, they do so in a way to minimise the impact to the tool’s users. Our accessors and mutators therefore implement a rule set on how we allow communication between classes, creating the necessary expected behaviour to build.

**Inheritance/Abstract Class**

Inheritance allows us to remove repetition in our code. When we separate the concerns of the connect4 game into unique classes that handle their own operations, this doesn’t mean repeating common code. We created the Abstract Class GUI to act as a template for all of our GUI based classes to share common field and methods from a parent to its children.

The GUI is not instantiated in itself, but provides a mix of methods and fields that organise our subclass GUIs. We defined several non static methods used either publicly by other classes or protected to subclasses.

Firstly, all the GUI classes need to talk to the same game controller object. We created a game controller class field and a ‘setter’ method in the parent. This is inherited by any of the GUI child classes. The controller object can set itself as the GUI subclasses controller without having to write multiple setters and variables.

Secondly, the JFrame object is the main container in which our GUI components sit in. We don’t want separate container windows for our different GUI screens. Therefore, we define the size, name and instantiate the JFrame in the parent class for the subclasses to inherit. The subclasses now only have to add their own JPanels to the instantiated JFrame container. They don’t need to use a gettor/setter, because as subclasses they can modify the protected JFrame object directly.

Whilst the JFrame and game controller have fields, and we have non static method, we still wanted some quasi interface functionality in our abstract class. We created an abstract method called showScreen() and left the method body blank. Whenever we extend into a subclass that is to be instantiated, we created a hard rule that they must implement a method called showScreen(); and use polymorphism to override the abstract parent placeholder method. Simply, if you’re going to make a new screen class (JPanel to be precise) you need to make sure you write code to add it to the JFrame and that method is named in the same way.

We have several nested local classes in our code. The game win logic nested local subclass of the game controller was separated because it had a specific focus but several methods and nearly 100 lines of code. Whilst it is distinct from the controller parent class as its responsible just for checking who has won, it still handles/responds to player action. We want all the attributes of the game controller and its methods, without having to rewrite them.

**Interface**

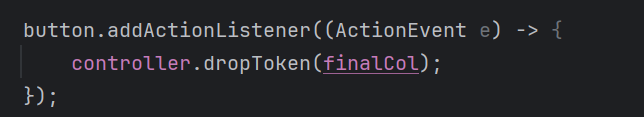
There are many interface examples through the connect4 project. We created an interface for the Player class to define the necessary methods to be implemented by any class that implements the interface. To be honest, most are placeholders, but getColour and getName are necessary getter methods that need to be created when the player interface is used. These are necessary because we can update the board colour based on who is the current player, and celebrate the name of the player who eventually wins.

We rely heavy on the ActionListener interface so that we can play our connect4 game. Our game screen is based primarily on one swing component class type the JButton. This class is a subclass of the abstract parent class AbstractButton. The abstract parent AbstractButton has the method addActionListener which the JButton inherits. We use this method on the JButton object to implement the ActionListener interface on itself.

ActionListener is a functional interface as it only has the one abstract method actionPerformed. This method requires as a parameter an object of class ActionEvent represented as ‘e’. When the JButton object implemented the ActionListener interface, it “registered to receive events” (Oracle 2024). This is reinterpreted as, when the JButton is clicked it triggers an ActionEvent because of the interface. That event would then be fed as a parameter to actionPerformed method which could have a body designed to respond to the event click.

Since actionPerformed method is from an interface, we need to implement it. But as actionPerformed is the only abstract method of the interface (making it functional), we can implement the method indirectly. We can do this when we used the ‘addActionListener’ method. We pass Class ActionEvent ‘e’ as a parameter to that method when we first add the interface to the JButton object. This will invoke for us the abstract method actionPerformed because it is of the correct parameter type without us needing to write it explicitly.

Finally, we can use a lambda as the body of the implicit actionPerformed method to respond to the event click.



Here we take the JButton object ‘button’, added the interface with the addActionListener method and passed the ActionEvent ‘e’ parameter. Instead of writing out the actionPerformed method body, we used -> to create a lambda body to define the response as the dropToken method of the controller Class. This ensures that when a button is pressed, a token is dropped.

The benefit to the connect4 game of this combination of interfaces and lambdas is more modular and concise code.

**Reference list:**

Oracle (2024) Class ActionEvent (Java Platform SE 8) available at: <https://docs.oracle.com/javase/8/docs/api/java/awt/event/ActionEvent.html> (Accessed 24 October 2024)