SYSC2004 Winter 2018

Assignment 4

Objectives:

- Build a GUI-driven ConnectFour game as an exercise in interfaces, abstract classes, the JavaFX Event Handling Pattern as well as the Observer Pattern.
- Hone problem-solving skills by developing a program with less guidance and more self-directed learning.

Provided: ConnectFourGameTest.java, ConnectFourEnum.java

Submit: ConnectFourGame.java, ConnectFourTestClient.java, ConnectFourApplication.java [with inner class ButtonHandler.java], ConnectButton.java, ConnectMove.java

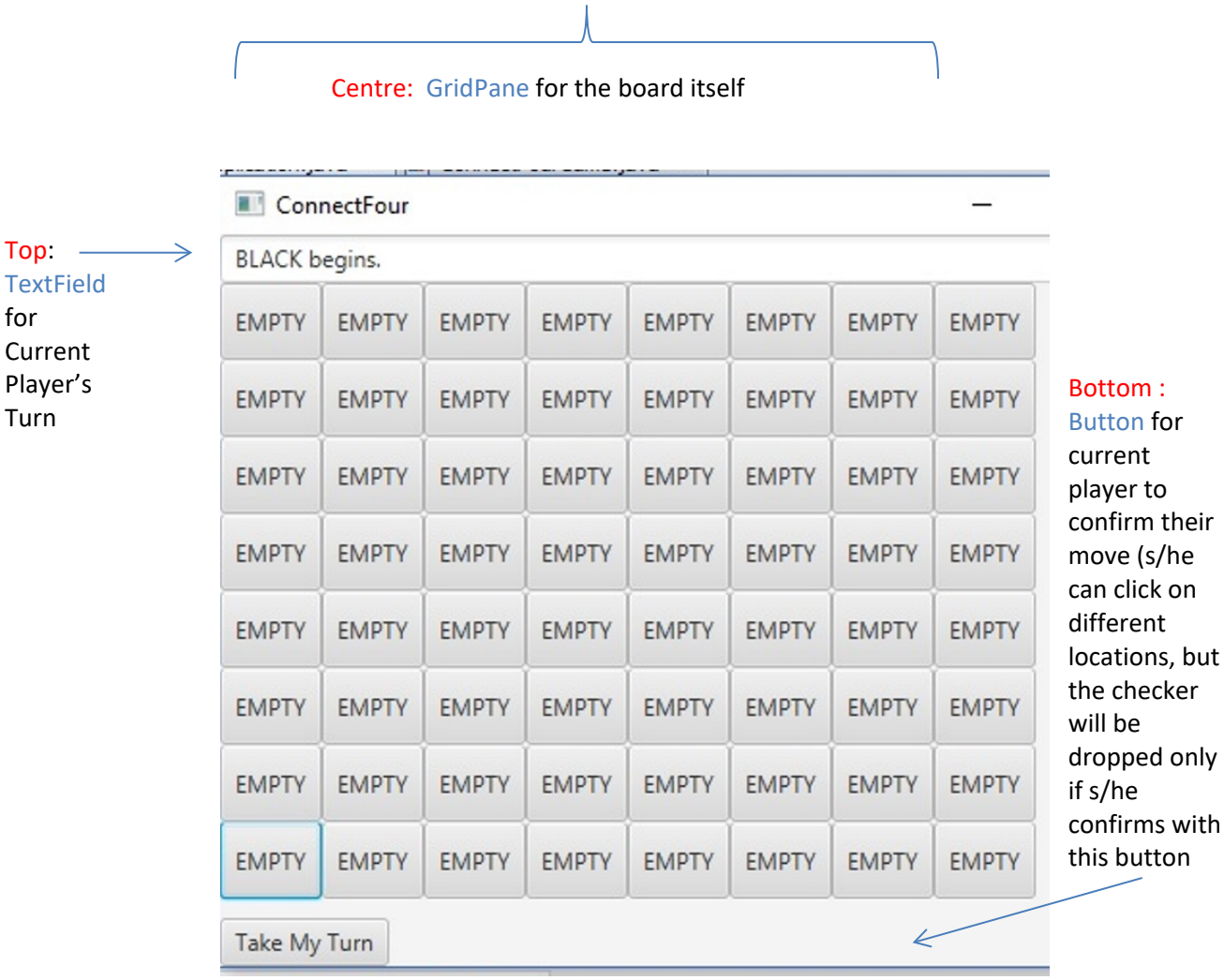
Background Information (Use as reference for the whole assignment)

Your task is to build a GUI that looks and feels like the game called **Connect-Four**. If you have never played Connect-Four, do some background research on the internet. Once you understand the gist of the game, here are our rules (some are adaptations that simplify your work).

- Nominally, the grid is an 8x8 matrix, but using good programming conventions, you will write your code so we can make it any size.
- Players take turns starting a game, the first player being randomly chosen. In each turn, one checker is dropped into an empty location that is either at the bottom of the grid or is above another checker [a checker cannot be dropped above another empty location].
- Players win by making a row – either vertically or horizontally – of 4 checkers of their own colour. Again, of course, you will write your code so that we can change the number of checkers needed to win.
    - Note that diagonal rows are not required. This is a simplification to reduce your workload. Interested students are encouraged to add that logic to their game.
- If the grid gets filled up with neither player forming a row of the required length, the game ends in a draw.
- After the game is over – either by a winner or in a draw – the game is reset and play continues.

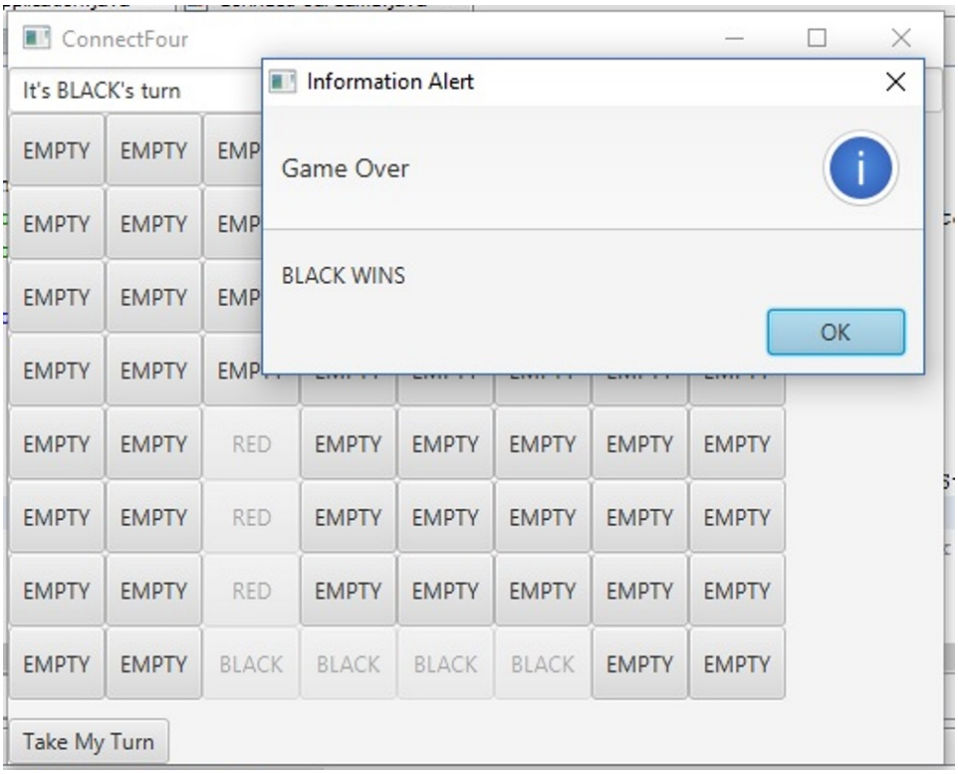
Our GUI-Driven Game

Below is the view that you will replicate in this assignment.



The `scene`'s root is a BorderPane (Do a Web search: Java API BorderPane).

- In the centre of this outer `BorderPane` is a GridPane (Do a Web search: Java API GridPane). The contents of the `GridPane` are `Button` objects. Initially, the buttons' labels are all EMPTY. As the players drop checkers, the buttons' labels are changed to RED or BLACK, respectively, and the button is disabled (See the second image below)
- The top of the `BorderPane` is a `TextField`, displaying which colour's turn it currently is.
- The bottom of the `BorderPane` is a `Button`, containing a button that confirms that a player wishes to drop a checker on a selected location. Clicking on buttons will not drop a checker until this button is clicked.



When the game is over, an INFORMATION ALERT dialog is displayed, saying who won the game (RED, BLACK or a DRAW). When closed by the user, the game is reset and play resumes with the next game.

The Complete UML for this Assignment

The UML class diagram below will be referenced throughout the remainder of this assignment. In contrast to other UML diagrams in this course, you are allowed to – **and will need to** - add variables and methods to those listed. Use good coding conventions for your choice of variables and methods.

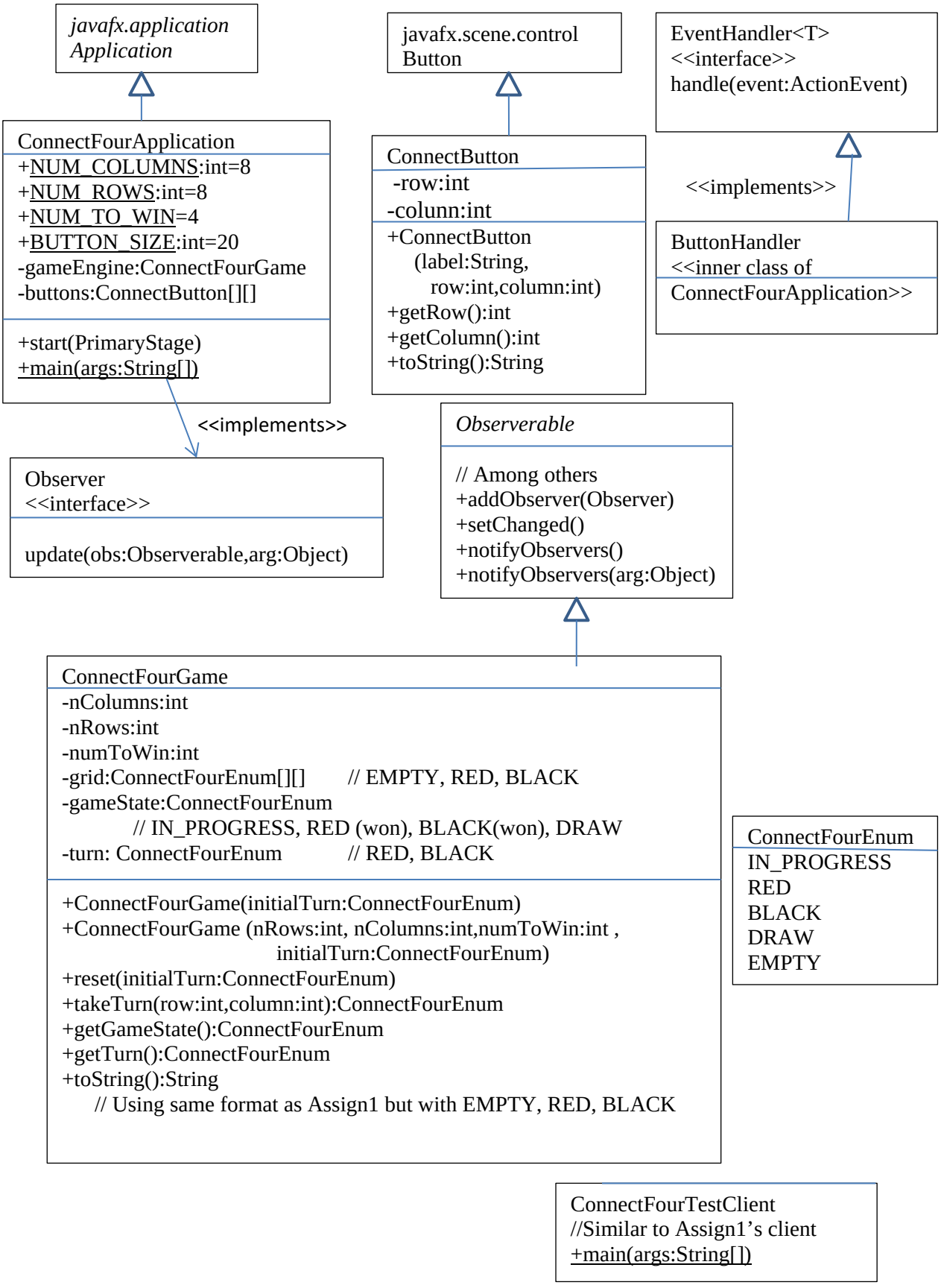
Read the UML in stages (A good practice to learn. Don't code yet)

Stage 1: To begin, ignore all the GUI-related classes and focus on the model class – `ConnectFourGame`. In doing so, answer this question: **What is the difference between ConnectFour and the TicTacToe game of Assignment 1**? Except for the size of the grid, these two games are very similar. One major functional difference is that checkers in ConnectFour must be dropped in locations from the bottom-up; you cannot put a checker in a grid location that is above an empty location.

- Because of the similarity, a big hint is to make use of your code from Assignment 1.
- If we had more time in this course, we would be considering the bigger design question: Could/should we use either inheritance or composition to re-use TicTacToe when writing ConnectFour. This is going to make a great class discussion.

Stage 2: Shift your focus to the GUI portion of the system. Notice the usual *javafx.application.Application* inheritance hierarchy for the MVC-view (`ConnectFourApplication`) and the parallel inheritance hierarchy of the MVC-controller (`ButtonHandler`). `ButtonHandler` is to be written as a named inner class of `ConnectFourApplication`. After completing this assignment, you should be prepared (on an exam) to answer why (as opposed to a public class). There is a third inheritance hierarchy involving the `ConnectButton` class. You will write this class as a subclass of *javax.scene.control.Button* to make the usual JavaFX buttons but also carry along some extra valuable state information (that may be handy when handling button clicks) You are seeing inheritance at work.

Stage 3: Finally observe the Observer pattern. The Observer pattern will be used to trigger updates to the View by relaying state changes in the model in response to button-clicks.


javafx.application
Application
javafx.scene.control
Button
EventHandler<T>
<<interface>>
handle(event:ActionEvent)
ConnectFourApplication
+NUM_COLUMNS:int=8
+NUM_ROWS:int=8
+NUM_TO_WIN=4
+BUTTON_SIZE:int=20
-gameEngine:ConnectFourGame
-buttons:ConnectButton[][]
+start(PrimaryStage)
+main(args:String[])
ConnectButton
-row:int
-column:int
+ConnectButton
(label:String,
row:int,column:int)
+getRow():int
+getColumn():int
+toString():String
<<implements>>
ButtonHandler
<<inner class of
ConnectFourApplication>>
<<implements>>
Observer
<<interface>>
update(obs:Observerable,arg:Object)
Observerable
// Among others
+addObserver(Observer)
+setChanged()
+notifyObservers()
+notifyObservers(arg:Object)
ConnectFourGame
-nColumns:int
-nRows:int
-numToWin:int
-grid:ConnectFourEnum[][] // EMPTY, RED, BLACK
-gameState:ConnectFourEnum
// IN_PROGRESS, RED (won), BLACK(won), DRAW
-turn: ConnectFourEnum // RED, BLACK
+ConnectFourGame(initialTurn:ConnectFourEnum)
+ConnectFourGame (nRows:int, nColumns:int,numToWin:int ,
initialTurn:ConnectFourEnum)
+reset(initialTurn:ConnectFourEnum)
+takeTurn(row:int,column:int):ConnectFourEnum
+getGameState():ConnectFourEnum
+getTurn():ConnectFourEnum
+toString():String
// Using same format as Assign1 but with EMPTY, RED, BLACK
ConnectFourEnum
IN_PROGRESS
RED
BLACK
DRAW
EMPTY
ConnectFourTestClient
//Similar to Assign1's client
+main(args:String[])

Project Preparations

You need a project that is both a new JavaFX project (not a Java project).

Part 1: ConnectFourGame – The Model

This part is essentially a repetition of Assignment 1. Go back to Assignment 1 if you have trouble remembering the steps yourself. Take a moment to reflect upon how much you have learned from the beginning of this course.

Implement the `ConnectFourGame` class along with the `ConnectFourEnum` class. Remember to make the few changes needed:

- A checker cannot be put above an empty location (it will fall down!). An `IllegalArgumentException` must be thrown if this is attempted.
- Consequently, when looking for a winner, your search need only start with respect to the location of the most recent checker (whereas in TicTacToe you always had to check the entire grid).

You are required to **test** this class without a GUI.

1) You are provided a standard unit test `ConnectFourGameTest`.java. It does some minimal testing of your class that are not comprehensive but will ensure that you are on the correct path.
    - If you read the tests, you will notice that the `toString`() tests are commented-out. I don't want you to spend time getting an exact match of the strings. The important aspects to test are the `gameState`, whose turn it is, and whether an exception is thrown if a checker is placed above an empty location.
2) You must write a test client `ConnectFourTestClient` that plays a game using console text input-output. This class contains solely a `main`() method. This `main`() method is almost identical to the client test code that was provided for you in Assignment 1, for `TicTacToe`.
    - There is a good reason for this small task, related to the Model-View-Controller pattern. The console text input-output is simply one VIEW. The GUI is a second VIEW. The model will work with ANY / MULTIPLE views. This is the benefit of the MVC pattern.

Part 2: ConnectButton – Preparing for the View

Now we get to move onto the fun GUI stuff.

Implement the `ConnectButton` class as shown in the UML. It is a simple class and the implementation should be thought of solely as an exercise in inheritance. The `toString`() method should return the following format: "**(**<row>**,**<column>**)**". It will be very useful in the next part.

Part 3: ChessBoard GUI (The View and a Minimal Controller)

Suggestion: DO NOT ADD THE OBSERVER PATTERN YET

A common error will likely happen to most of you, and here's a suggestion to avoid it : Use incremental software development to find small bugs early, rather than big bugs later.

1. Write the `start`() method of `ConnectFourApplication`. In this method, create all the `ConnectButtons`, the `Textfield` and the confirmation `Button`. Forget about event handling until your GUI looks good.
    - There are some handy methods inherited from the `Button` class for achieving a regular look to your grid.
        - `setMinHeight`(20) – Makes your button a big taller (to get a squarer grid)
        - `setMaxWidth`(`Double.MAX_VALUE`) – Makes your button fill the space horizontally. Later, when you change the label to RED and/or BLACK, your buttons will remain a consistent width.
    - Using the two above settings, the dimensions of my `Scene` are initialized to 510 x 380
2. Implement the `ButtonHandler` class, but make it a minimal implementation: Simply print out the source of the event. Register this handler with each button.
3. Run the program and carefully pay attention to the messages printed when you click on different squares. Is (0,0) where you expect it to be? If not, go back to Step 1 and re-order your grid of buttons so that (0,0) is on the bottom-left corner.
4. You must also implement (and register) a separate EventHandler for the "Take My Turn" button. Because there is only one such button, you will implement this EventHandler as an Anonymous Inner Class. As a first version, again, simply print out a helpful debugging message, such as "Drop the Checker". In this first step, you want to simply verify that clicking on "Take My Turn" makes this piece of code run.
5. Only once you have correctly established the orientation of buttons in your grid, you can now write the full implementation of the `ButtonHandler`. According to MVC, `ButtonHandler` should update the model appropriately by dropping the checker in the clicked-on location.
    - Don't expect your GUI to update in response, yet. That's in the next part. If you have troubles, read onto the next part for more detailed help.

Part 4: Observer Pattern … Finally!

Implementation of the Observer Pattern will close the loop and cause your GUI to be updated, to show checkers being added to the grid after each turn is played.

- Refer back to the UML class diagram to identify the `Observable` and the `Observer`
- Remember to subscribe(or register) the `Observer` to the `Observable`. Nothing will happen if you forget this step.
- Remember that the `Observable` must call `setChanged`() before calling `notifyObservers`()
    - Both are done when some instance variable changes. In our game, this happens **when a checker is dropped successfully.**
- Remember that there are two overloaded versions of the Observable's notifyObservers() method
    - `notifyObservers`() – has no argument.
        - When the Observer receives notification on its `update`() method, its Object argument is null
    - `notifyObservers`(Object arg) – has one argument that the `Observable` can use to pass extra information to the `Observer`.
        - When the Observer receives notification on its `update`() method, its Object argument contains this extra information.
    - It is suggested that a `ConnectMove` object be passed to the `Observer`. The role of the `Observer` is to update the screen to show the latest checker dropped. It needs to know where the checker was dropped, as well as its colour. `ConnectMove` will help.

ConnectMove
-row:int -column:int -colour:ConnectFourEnum
+ConnectMove(row:int,column:int, colour:ConnectFourEnum) +getRow():int +getColumn():int +getColour():ConnectFourEnum

At this point, you can now play your game, watching the GUI update as you drop checkers onto location. If all is working fine, your final task to complete is to add in an ALERT INFORMATION dialog upon completion of the game. (Continued next page)

Tip: Instead of `Alert.show()`, try using `Alert.showAndWait()`. This will pause your game until after the OK button is clicked. In this way, you will be able to see the winning play before the grid is reset for the next round.

Marking Scheme (Total : 56 – 14*4)

Assignments will not be accepted if standard indentation is not used. A mark of zero will be earned.

Instruction to TAs: Make sure your write your initials on each assignment that you mark, so that the student knows who to contact for concerns about their mark.

Instructions to Students: Afterwards, if you have a question about the marking of your assignment -

1. On the course webpage, study the schedule for the TAs and see when your TA is on duty in the lab. If you do not have a class/lab conflict with that time, visit the TA during their lab period.
2. Alternatively, email the TA that marked your assignment. Pose your question if it is simple; if not, request a meeting. If you do not receive a reply from the TA within 2 working days, re-send your email and now 'cc your instructor so that s/he can follow up.

Marking Values: Each criterion is worth 4 marks and will be marked according to the rubric levels of BDAE (0 is not present at all, or consisting only of the UML call signatures)

Beginning for 1 mark: A bit but mostly not.

Developing for 2 marks: Some but missing key points and/or ignoring requirements & conventions

- Maximum mark possible if the code does not work.

Accomplished for 3 marks: Mostly of the vital learning points demonstrate mastery

Exemplary for 4 marks: Professional level code that is even better than asked.

Part 1 – The Model

/4 ConnectFourGame passes the minimal unit tests provided, and the code itself is correct

/4 ConnectFourGame uses exceptions to validate all arguments related to the size of the board

/4 ConnectFourGame's takeTurn() method is well-structured and easy-to-read, perhaps using a private function to encapsulate the complicated logic of looking for a winning-run. This logic looks ONLY at sequences BELOW the latest checker added.

/4*2 ConnectFourTestClient – A simple text-based client allows testing of ConnectFour by playing an actual game. Client successfully runs, and has minimal and clean output, showing the current state of the board, and repeatedly prompting the user to take another turn, until the game is won.

Part 2 – ConnectButton

/4 The class is correctly implemented as described by the UML class diagram, and it is well-constructed using conventional OO practices.

Part 3 – Basic GUI

/4 Look - When run, the GUI has the described organization: Top, Bottom and Centre. The Centre is a grid of Buttons.

- Exemplary – Must have regular, pleasingly-sized EMPTY buttons as well as the current player displayed in the Top's textfield. It should look like a grid, and the grid must match the

/4 Feel, part 1 - The ButtonHandler is correctly implemented as described by the UML class diagram, and as a non-public class. When run and when the user clicks on a square, this ButtonHandler is notified.

- N.B. If the whole assignment is complete and the game works, this mark is automatic

/4 Feel, part 2 - The “Take My Turn” button correctly invokes an separate event-handler when it is clicked.

- Exemplary: Requires that this event handler is implemented as an anonymous inner class.
- N.B. If the whole assignment is complete and the game works, this mark is automatic

Part 4 Observer

/4 When run, the program lets you play a correct game of Connect Four (i.e. can only drop checkers above filled spaces, and winning runs are found).

/4 Code Inspection: The Observer Pattern is correctly implement: Observable is subclasses, Observer is implemented, and the Observer adds itself to the Observable upon startup

/4 Code Inspection: The ConnectMove class is implemented as described

/4 Code Inspection: The Observable updates only the model, and the model notifies the Observers to update the view.

/4 An Alert dialog is used to signal the end of the game.

Overall Code Inspection:

DOES the code adhere to the Java Style Guidlines? TBD (https://learn.zybooks.com/zybook/CARLETONSYSC2004SchrammWinter2018/chapter/2/section/8) . /4 - For Whitespace – Appropriate, consistent whitespace between lines of methods as well as spaces between operators and arguments (Example: x = 5; instead of x=5;). Readability.

/4 - For Braces – consistent placement of { } for readability

/4 - Naming of any working variables using conventions

/4 – Efficient use of variables – avoidance of repetitive allocation of variables through judicious use of variable scope.

/4 – Well-constructed JavaDoc for all methods in all classes.