**Offline marking (without a demo) submission document**

If you want to submit your work for offline marking rather than demo then please fill in this form and submit the following to the ‘offline marking’ coursework submission box by 11am on Thursday 5th May (the end of the lab) so that you don’t get an advantage from doing this. Your submission time matters – ensure you submit by the end of the lab!

**Submit the following:**

* A zip file of your code (the .java files, e.g. from the src directory), with the directory structure so that we can unzip it and compile it with no problem. Zipping up your src directory from Eclipse, while keeping the directory structure will satisfy this.
* Your class file as a pdf. Note: this MUST be a pdf if you want to do offline marking – not any other format, so that we can read it. (For demos it didn’t matter because you will show us on the screen.)
* This file, with the information below complete.

**Main class: tell us what the main class is that has your main() function in it.**

Main.java -> Main

**Relationships: explain your key relationships from your class diagram – it helps us if you got something wrong – by filling in the classnames <> below for two example relationships in your diagram:**

<Main> HAS -A <boolean>

<Window> HAS - A <Layout> Composition (comprised of)

<Window> HAS - A <Int> Composition (comprised of)

<Window> HAS - A <JFrame> Composition (comprised of) (in hindsight I should’ve inherited JFrame)

<Clickable > IS – A <JButton>

<Clickable > HAS - A <Color> Composition (comprised of)

<Clickable > HAS - A <int[]> Composition (comprised of)

<Clickable > HAS - A <int> Composition (comprised of)

<Notification> IS – A <JFrame>

<Layout> IS – A <Jpanel>

<Layout> HAS - A <Window> Aggregation (references to)

<Layout> HAS – A <Clickable> Composition (comprised of)

<Layout> HAS - A <int> Composition (comprised of)

<Layout> HAS - A <ArrayList> Composition (comprised of)

<Layout> HAS - A <GridBagConstraints> Composition (comprised of)

<Layout> HAS - A <int[]> Composition (comprised of)

<Layout> HAS - A <Color> Composition (comprised of)

<Layout> HAS - A <boolean> Composition (comprised of)

We won’t have the opportunity to ask you as we would in a demo, so by giving us the two examples above we can see whether you understood properly.

**Where in your code did you check for the case where neither player can make a move and you therefore end the game and say who won:**

<Put your answer here – explain the file and line numbers and roughly how you check – e.g. how do you know if one player can’t play and how do you check for that applying to both players?>

In the function at line 229 in file Layout.java there’s a function named endgame(). The purpose of this function is to detect if the game has ended after a turn. The function is called on lines, 225,219. The function only checks if all the discs are one colour or if all of the discs have an active state, it doesn’t check if both players can’t make a move. The function first initialises variables **none, p1 and p2** for counting the blank discs, player 1 discs and player 2 discs respectively. It then decides a winner with some ternary logic (even if no winner is possible at this point, it’s to check who’s won if the end game happens on this turn). It then checks if all the tiles are active or if one of the players has no discs on the board, hence the full board and one player only having discs, determining a win. From there, a Notification object is created which when closed, closes all the windows and terminates the program.

**Which marks did you expect to get: (this is so that we know if you didn’t implement some things, rather than us trying to get things to work that you didn’t do). Please go through the list on the next page and put a Y against any marks you expected to get. If you don’t add a Y we will not test that mark.**

**Marks you expect to get:**

|  |  |
| --- | --- |
|  | **5 marks: class diagram is correct and represents the code.** |
| Y | • All of student’s own classes are shown, as are parent classes for these classes. Classes are shown correctly. |
| Y | • Attributes are listed for your classes (ignore standard classes). |
| Y | • Methods are shown for your classes (ignore standard classes). |
| Y | • Relationships between objects are correct. |
| Y | • Includes inheritance and aggregation correctly. |
|  |  |
|  | **9 marks: user interface is correct:** |
| Y | • Two top-level windows appear and work. (Three sections per window – label, grid, button.) |
| (appears in title instead) | • Each frame has as its top section a label which shows which player this board is for and whether it is their turn. |
| Y | • Each frame has middle section as an 8x8 grid of squares. |
| Y | • Each square in grid is green with a black border. |
| Y | • Each square with a piece has either a black circle with white outline or white circle with black outline on them. |
| Y | • Bottom section of each board/frame is a button for asking AI to make a move – see marks below. |
| Y | • Boards are represented from the point of view of different players – i.e. one is a 180 degree rotation of the other. |
| Y | • At the end a message is displayed which shows which player won (or a draw) and correctly counts how many pieces each side had. |
| Y | • There are no problems at all on user interface/usage – it all works correctly and appears correctly. |
|  |  |
|  | **7 marks: Basic functionality was implemented** |
| Y | • Only the active player can make a move. (Requirement above specifies that labels need to show which is the active player.) |
|  | • If there is no valid position for the player to make a move, control automatically moves back to the other player. |
| Y | • The active player can click any valid square on the board (where opposing pieces could be captured) to place their counter and the counter appears (or could use the AI - see below for AI). |
| Y | • Counters can only be placed in valid locations – when at least one opposing counter would be captured by the move. |
| Y | • When a counter is placed, control goes to the other player. |
| Y | • Placing a piece correctly captures (at least some) pieces between it and other friendly pieces. |
|  | • ALL appropriate pieces in all directions are captured (including horizontal, vertical or diagonal). |
|  |  |
|  | **5 marks: Board state is maintained** |
| Y | • The board state is correctly stored in a board object (or squares of a board object), so the state (white, black or empty) of each square on the board is maintained. |
| Y | • The correct pieces are shown on both boards for the current board state. |
| Y | • The initial board state is an alternating white (top-left), black (top-right), black (bottom-left), white (bottom-right) pattern in the middle 4 squares of the board. |
| Y | • When the board is filled, the game will end. (Message will be displayed – see above.) |
|  | • After completing a game, a new game can be played, with an initial board which is set up correctly, and the game can progress correctly. |
|  |  |
|  | **4 marks: Simple Artificial Intelligence** |
| Y | • When the button is pressed the AI makes a decision about where to place a piece |
|  | • AI correctly counts how many pieces would be captured if it made a move into each of the empty squares. |
|  | • AI selects one of the square(s) which would capture the highest number of pieces and makes a move there. |
|  | • AI makes a move correctly, actually setting the board square. Both frames are updated correctly to show the move. |