COMP1009: Programming Paradigms

Object Oriented Coursework 2c

You must complete this coursework on your own, rather than working with anybody else. This is not a coursework to do in pairs.

To complete the coursework you must create a working two-player version of a Reversi game (https://en.wikipedia.org/wiki/Reversi), along with class and object diagrams to explain your chosen design.

This coursework is harder and larger than the previous courseworks and lab exercises, but the previous exercises should have given you the chance to practice the necessary skills.

(Repeat) You must complete this coursework on your own rather than working with anybody else. All of the work you demo/submit must be your own, and by demoing/submitting your work you verify this.

This coursework is worth 12% of the module mark. You should be able to get full marks if you work through it systematically. You are suggested to consider what you did in the earlier labs and courseworks to work out how to implement different elements. Lab 4 will be especially useful for the GUI side.

You are recommended to try to complete this coursework in your own time and to use the labs as an opportunity to get questions answered. The lab helpers can help you to understand concepts but will not help you to actually complete the coursework itself.

Be aware that this coursework deliberately gives you less of a walk-through than the previous lab exercises, although it does give you various hints. It usually tells you what to achieve, rather than how to achieve it. Also, you may want to implement things in a different order.

Against each point in the requirements you can see a number of points to check. These are things that I may test when I mark this, so it is worth checking each one to ensure that you didn't miss anything. I am happy to give you all full marks if your implementations work.

I will test your controller and GUI separately. This means that you can still get marks for one of them even if you don't complete the other – as long as it works properly. Also, since these are independent, you could decide to implement one first and then the other. Note that I suspect that the GUI is easier than the controller, because the GUI just shows what you tell it but the controller makes decisions. Implementing the GUI first may help with the controller.

The general requirements give you an overview of what to do and what object oriented things to consider. The program requirements give you an idea of what the program needs to so, in terms of features, and what to check that you implemented.

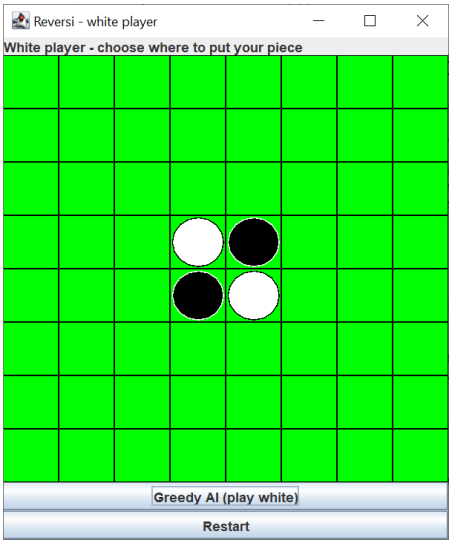
If you did not attend the lecture which introduced this coursework, then watch the video before attempting this coursework – it should make it a lot clearer and give a lot of hints.

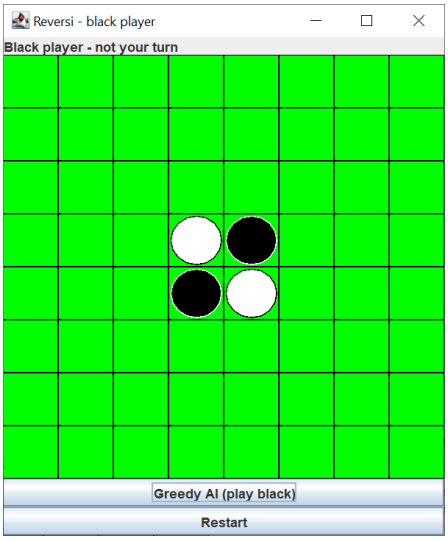
There are three things to consider: the general requirements (general things that you need to do), the program requirements (what your GUI and controller must do for you to get the marks), and the things to check afterwards. I suggest first implement the requirements then do all of the checks once you think it is finished. You can also test your GUI and controller with the various implementations that I have provided for the other components.

General coursework requirements:

Considering the following program requirements as well as the marking criteria:

- Produce a Java program for a Reversi/Othello game which meets the program requirements and marking criteria specified in this document.
- Only your main() function can be static. You must have no other static member functions! (Not doing this will penalise marks!)
- You have been provided with a number of classes and you must use these in your submission. This is important, so that the controller and GUI are separate from each other and I can give marks for one even if the other goes wrong. This should help you understand MVC, but it will also greatly aid testing of your program.
- You are going to implement a **class called GUIView which is a subclass of IView** for the GUI. You must use this class name so that the testing finds your class. Your view must work with the unmodified SimpleModel and SimpleController. To make everything work you need to treat this as an IView object, and only access the methods of IView from the controller or model. (See program requirement 1.)
- You are going to implement a **class called ReversiController which is a subclass of IController** for the controller. You must use this class name so that the testing finds your class. Your class must work with the SimpleController, unmodified. To make everything work you need to treat this as an IController object, and only access the methods of IController from the controller or model. (See program requirement 1.)
- You should be able to choose any combination of controller and view in ReversiMain and it should still work. This means that you can use SimpleController to test your view and TextView to test your controller if you want to implement them one at a time – experiment with these.
- Use ONLY standard swing/awt GUI classes, the provided classes and your own classes (which may be subclasses of Swing classes). You may also use collection classes (e.g. ArrayList) if you wish, although I didn't need to do so. Don't try to access the file system or network, as doing so will cause your program to end. If you need other classes then please ask in the "OO and Java Questions" channel on the COMP1009 team on MS teams, saying why. This will allow everyone to have the same information and means that if I allow one person then everyone else will have permission too.
- You will need to understand how to create objects, and how to have references to other objects within your objects. This is a key thing to understand.
- Your finished program should look like the pictures on the right at the start (i.e. it has 4 pieces already played), should allow players to place pieces when it is their turn, should automatically change the colour of pieces which are taken, and should detect when a player wins, counting the pieces of each colour to determine the winner. You will also implement a simple 'greedy' AI.





Program requirements:

1. **GUI and Controller:** You will implement the GUI and controller to allow you to play Othello/Reversi on the computer (see General Requirements). If the rules of reversi are not clear from the coursework introduction lecture, please read the information about how to play Othello/Reversi first.
   Wikipedia has a summary of the rules: https://en.wikipedia.org/wiki/Reversi

   You will need to have a view class called GUIView which appropriately implements all methods of IView, to initialise (store object references), refresh the appearance according to the data in the model, and set labels appropriately. Check that each of these three functions works correctly – especially refresh().

   You will also need to have a controller class called ReversiController which will implement all of the methods of IController appropriately – to initialise (store object references), startup (start or restart the game), update the status if something has changed, handle a square being selected and implement a greedy AI move. Check that each of these five functions works correctly.

   All of your files will be in a package called reversi (lower case).

   If you got the functions names correct then you should be able to just change which lines are commented out in the ReversiMain.java file. This is a good check of the naming.

   Check: your view class has the correct name and implements all three functions correctly.
   Check: your controller class has the correct name and implements all five functions correctly.
   Check: the update() function works properly on your controller, so that if something is changed in the model and update() is called, your controller still works properly. Ideally this is easy as long as your controller is not storing any information. Hint: the SimpleTestModel is an easy way to check this, since testing is its main purpose.
   Check: the refreshView() function works properly on your view, so that if something is changed on the board and refreshView() is called, your view shows the correct contents.

   Note: You may have other classes as well if you wish, that these classes use (e.g. a class for a square on the board), but you need these two classes so that I can test it.

2. **GUI:** You must have two frames which display (different) views of the board. The frame title for each should specify which player the board is for.
   There should be two buttons at the bottom – one for running a 'Greedy AI' and one to restart the game – clearing the board back to the first position.
   Hint: I put a label in the North position, a grid layout in the Center for the board, and another grid layout in the South position, to hold the two buttons.
   Check: 2 frames. Label, board and 2 buttons in each frame.

3. **GUI:** The boards shown in the frames should be from the perspective of the two players – so one should be a 180 degree rotation of the other (i.e. the top/bottom and left/right are both swapped). Hint: on one board number the x from 0 to 7, left to right, and the y from 0 to 7, top to bottom, so the top left is (0,0). On the other board number the x from 7 to

0, left to right, and the y from 7 to 0, top to bottom, so the top left is (7,7).
Hint: If you use a grid layout this just means adding the labels/buttons that you use for the squares in a different order.
Think of each as frame representing the view of the board for one player, and the players are sitting at opposite ends of the table, so that they see the board upside down (actually rotated 180 degrees) compared to the other player.
Note: you will need separate labels/buttons for each frame – it won't work properly if you try to add the same label/button object to two frames.
Check: boards are shown from different viewpoints, rotated with respect to each other.

4. **GUI:** Create a class to draw and handle a square of the board. Consider what I did previously in ColorLabel and do something appropriate for the board square – probably a button rather than a label though. You will need to draw a green square, a black border, and white/black circle with a border in the opposite colour.
   Draw the contents yourself (do the same as I did in ColorLabel, just with rectangles and ovals) as it will give you practice overriding a function and implementing some drawing.
   Hint: I used 50x50 squares, 46x46 ovals and did a fillOval in the colour I wanted, then a drawOval in the other colour to draw the border.
   Note: do not try to use images because it won't work when I test it, so I will spot this really easily, because you won't have access to load the image from the file system when I run it.

   You may want to watch lecture 10 on inner classes before deciding how to handle the selection of the square, as it gives you more options, or you could do a button which notifies your button object itself when it is pressed (i.e., it is its own listener – which sounds more complicated than it is). Lecture 13 on anonymous classes and Lambda expressions gives even more options but it unnecessary so don't delay starting this – I just made my buttons their own listeners.

   Hint: Your labels/buttons for your squares will need to be able to tell the controller when they are pressed, which means that they need a reference to the controller. The easiest solution in my opinion is to pass the object reference to the controller into the constructor when the object is created, and store it for later use. You can implement this however you wish, though. (This is quite an important concept to understand.)
   Similarly, they will also need a reference to the model, to be able to work out what piece is in the square at the moment (the model stores this information for you). Do not try to remember in the button/label what it should contain – that's the model's job – and will break other things if you do – e.g. when the controller tries to reset the board.

   Check: You have a class to handle a board square.
   Check: Contents of square depend upon data in the model – so if the model changes and the controller refreshes the view, the appearance of the buttons will change.
   Check: When a square is selected by user, the controller is told (by calling the method).

5. **GUI and Controller**: Both the GUI and controller should use the supplied SimpleModel to store your board state, who the current player is and whether the game has finished or not. They should not store this information themselves.
   Optionally you can test with its subclass, called SimpleTestModel, which does exactly the same thing but gives you some test buttons you can press.
   Look at how the model is used by the existing views and controllers to understand how to

use it yourself.
Your controller should put four initial pieces in the centre as show in the pictures on the previous page.
Check: Initial board has the correct pieces in the centre.
Check: Model stores all of the data. E.g. there is no tracking of the board state, who is the current player, or of whether the game has finished, anywhere else than in the model.
Check: Model is unchanged from the initial version supplier. This is important as I will use the original one even if you change it!

6. **GUI:** When a square is clicked the GUI should tell the controller and the controller should decide whether to put a piece in the square or not. i.e., the GUI should not make any decisions about whether a move is valid or not – that is the controller's job.
   Check: Controller is told when a square is clicked.
   Check: Model stores all of the data about what each square should show. i.e. if the data in the model is changed and someone calls refresh() on the view, then the buttons will show the contents according to the model.

7. **Controller:** The controller will decide what to do if a player selects a square:
   - If there is no valid location to play in, then change to the other player. See point 10.
   - If neither player can move then end the game, telling both players the final score and ignoring any more input until one player presses restart. See Point 11.
   - If it is not that player's turn then inform that player by setting the feedback message to "It is not your turn!" (Get the text exactly right please so I can test it, see point 16.)
   - If it is an invalid location by the correct player then ignore the click – do not send a feedback message, so that it still shows the message to choose where to put their piece.
   - It is only a valid location to play if it can capture one or more of the opposing pieces. In which case, put the piece in, perform the capture (see point 8) and change to the other player's turn, sending appropriate feedback messages to the GUI.

   Check: Controller rejects playing a piece by current player in an invalid square.
   Check: Controller rejects the wrong player playing a piece – and sends feedback to player.
   Check: Controller goes to the other player if one player cannot move (see point 10).
   Check: Controller ends the game if neither player can move (see point 11).

   Hint: You will need a function on the controller to work out how many pieces would be captured if a player plays in a specific location. This will be used by the greedy AI (later) but can also be used to work out whether a square is a valid location. If the number of pieces captured is zero then the player cannot play a piece there. Please also see the hints on the last page of this document.

8. **Controller:** When a piece is played, any pieces between it and another piece of their own colour in a straight line up to the first piece of your colour in that direction are captured. This is probably the hardest bit to implement, so give it some thought.

   Hint: One way to do this is: every time a piece is played, search vertically, horizontally and diagonally (8 directions in total):
   - If you find an empty square or the edge of the board then stop – no pieces can be captured in that direction.

- If you find a piece of the opposing colour then keep a count of the number of these pieces you pass and continue in that direction, checking the next square.
- If you find a piece of your own colour then stop. If you have already crossed any opposing pieces (this is why you kept a count, above) then you would capture all of the pieces of the opposing colour between this piece and the piece you just played.
- If no opposing pieces would be captured in <u>any direction</u> then you cannot play there.
- If you do play there, then <u>all</u> pieces that could be captured in <u>any direction</u> are captured at the same time.

Hint: Please see the hints on the last page of this document.

Check: when you play a piece, all pieces between it and another of your pieces are captured.
Check: Capture can happen in multiple directions at the same time. Theoretically in all 8 directions at the same time.

9. **Controller:** White should play first. Players then take it in turns to play. (Note points 10 and 11 though.)
   The active player should be made obvious to the player(s) and should be indicated in the label along the top of the board, as shown in the earlier pictures (where it was white's turn).
   Check: after playing a piece it becomes the other player's turn.
   Check: The correct feedback messages are sent to both players – telling one it is their turn and the other that it is not their turn – see point 16.
   Check: White plays first, even after a restart – i.e. restart sets the player back to 0.

10. **Controller:** If a player cannot make a move then the game switches back to the other player's turn automatically - the other player can make another move, unless the game has ended (see point 11).
    Check: if a player cannot play a move – i.e. there is no square where they could capture an opposing piece, then the other player becomes the active player and gets the feedback messages to tell them so.

11. **Controller:** When neither player can play a piece then the game ends (this will usually be when the board is full, but it may not always be full at the end). At this point the players should be presented with a message each, telling them then who won (who had the most pieces) and how many pieces of each colour there were at the time the game ended.
    Check: where there is no valid play location for either player then it ends, sending the correct feedback to each player, and preventing any player from playing a piece. Testing note: I can see whether you send the correct feedback messages, and can check that you no longer allow pieces to be added.

12. **GUI:** There is a button labelled 'Restart' on each frame, which when clicked will tell the controller to startup() again.
    Check: see point 13. Add some pieces and check that restart goes back to the starting position.

13. **Controller:** The implementation of the startup() method, which is called at the start or when the player asks to restart, will clear the board and put the initial pieces back into the centre.
    Check: see point 12.

14. **GUI:** There are buttons labelled "Greedy AI (play white)" and "Greedy AI (play black)" on the appropriate frames, in the bottom section, which when clicked will ask the controller to 'doAutomatedMove()' for the appropriate player number.
    Check: see point 15 as well. Press the button and check that it puts a piece in an appropriate position. You will want to try some cases where it can capture multiple pieces, and just check that the number it catches in each case is the most that could be captured. It is also useful to check that it can capture in multiple directions at once.

15. **Controller:** The implementation of the automated move will implement a simple greedy AI, so that when a player presses the AI button and it is their turn the computer takes their turn for them.
    The controller should check each space (via the model), calculate how many pieces will be taken if a piece is placed in that space and choose one of the spaces which take the most pieces. Note: If there are multiple places which take the same pieces then I don't mind which you choose – I chose randomly but you can choose the first or last such position if you wish. This is called a greedy algorithm because it always makes the best move it can think of at the time without thinking about later consequences. Please do read the hints!
    Check: see point 14.

16. **Controller:** You need to match exactly this wording for the feedback messages please, so that the testing will give you the appropriate marks for the feedback messages:
    - When it is the white player's turn, set their message to: "White player – choose where to put your piece".
    - When it is the black player's turn, set their message to "Black player – choose where to put your piece".
    - When it is not their turn, set the white player's message to: "White player – not your turn"
    - When it is not their turn, set the black player's message to: "Black player – not your turn"
    - If either player tries to make a move when it is not their turn, set the feedback message to: "It is not your turn!"
    - When the game ends, set each player's message depending upon who won, showing the number of pieces won by each player:
        - If black won (black had more pieces at the end), set the messages to "Black won. Black 33 to White 31. Reset the game to replay." Where you will change the 33 to show the actual number of pieces that black had on the board and the 31 to the actual number of pieces that white had on the board.
        - If white won (white had more pieces at the end), set the messages to "White won. White 33 to Black 31. Reset the game to replay." Where you will change the 33 to show the actual number of pieces that white had on the board and the 31 to the actual number of pieces that black had on the board.
        - If it was a draw (equal number of pieces at the end) then set the message to say: "Draw. Both players ended with 32 pieces. Reset game to replay.", replacing the 32 by the number of pieces that they both ended with.

      Note: testing of this requirement will check the piece count that you report.
    - There are no other feedback messages to send.

    Check: check each of the cases above and verify that the labels show the correct message.

17. **Controller and GUI:** Your controller and view should be independent from each other and from the model.
    Check: Add all of the original code back into your project, to ensure that you didn't accidentally change anything, then…
    Check: Test your controller with the alternative views that I provided.
    Check: Test your view with the alternative controller, which allows you to put a piece anywhere. Feel free to develop a test view or test controller to test other features – it will be good practice for you but if so, don't submit the files for marking.

18. **Controller and GUI:** Only the model should store the information about the board state. I am repeating this because it is important. Doing this it means that changes can be made to the contents of the model in testing and your controller and view will work properly.
    Hint: use model.getPlayer() to access the current player number, and model.hasFinished() to see whether the game has finished.
    Check: You should not have variables in your controller or view for the player number, whether the game has finished, or the board data.
    Check: It should be possible for a test system to change the data that is in the model (e.g. to change what pieces are on the board, which player is active, or to set the finished flag), then to call refreshView() on the view (to update the display of what is in each board square) and update() on the controller and everything should work appropriately. As long as you make sure that you always ask the model for information about what to put in each square, about which is the active player and whether the game has ended, rather than storing it in the view or controller then it will work.

Additional Hints:

Counting the pieces taken: If you create a function which counts how many pieces will be taken if you play in a location, you will find that you can use this function in multiple places. If the value is 0 for all directions then you can't play there. If you call the function mentioned above for every empty square on the board you can work out whether a player can play at all. If neither player can play at all then the game should end. You can also work out what the best greedy move is for the AI very easily if you have developed this function (i.e. the one which takes the most pieces). i.e. this function is REALLY useful!

You should consider whether you want two separate functions (one to count the pieces which would be captured and another to actually do the capturing, see point 8), or whether this should be one function with a flag to say whether to capture or not. It's up to you – it will not affect your mark as long as it works.

Checking in multiple directions: If you want to check multiple directions, you could use a loop of offsets and either recursion (as seen in functional programming) or iteration (we can do either in Java). Recursion can cause a problem with stack space if you call the function thousands of times, but will be safe for here as you will only recurse up to 7 times maximum (moving one square across the board each time).
E.g. continuously adding -1x,-1y to a position moves diagonally, whereas adding 0x,1y to a position moves vertically. I used two loops: e.g. for (xoffset=-1; xoffset<=1;xoffset++) and similarly for y. You need to skip the case for 0,0 – as this is not a move (no change in either direction). Using this offset I checked all 8 directions without me having to hardcode multiple directions.
If this doesn't make sense to you then just hardcode the directions, but it is not as quick to code it that way, so you'll need to write more code, and you risk copy-paste typos.
Don't forget to check for reaching the edge of the board!
You can use iteration to keep moving to the edge, but I found that recursion gave me an easier function to create in this case and it may help you in comparing Haskell vs Java as well. You may find iteration easier until you fully understand recursion though – the comments from the FP lectures about different cases also apply to recursive functions in Java.