COMP1009: Programming Paradigms

Object Oriented Coursework 2c and 2d (v2025.2)

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

To complete the coursework you must create a working two-player version of a Reversi game (https://en.wikipedia.org/wiki/Reversi). You will create this in two parts, which will be submitted and tested separately. One part will be the user interface and one will be the game logics.

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. 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.

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

Together the two parts of this coursework are worth 17% of the module mark. You should be able to get full marks if you work through it systematically, although many people will make at least a minor mistake, or miss some features. Marking will consist of a set of tests which will be applied to your submission, so that you get each mark or not. 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.

Important: the deadline is after the holidays, but you need to sort out any problems before the holidays because there are no labs during the holidays, nor are there simple places to get answers for questions (and ideally you don't want to be working on courseworks when you should be resting).

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. Since these are independently marked, you could decide to implement one first and then the other. Note that I suspect that the GUI is easier to start with than the controller, if you don't want to work on them simultaneously, 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 then give you an idea of what the program needs to do, in terms of features, and what to check that you implemented.

Model-View-Controller summary comments

In this coursework you will be using the Model-View-Controller pattern. It is worth understanding the basics of this pattern to understand how your work will run.

There are three parts to the project:

Model: the data storage. All data that you need to maintain should be in here. This includes information about: what is on the board; whose turn it is; whether the game has ended.

View: this is responsible for showing the state of the game to the user, and accepting user input.

Controller: this is responsible for making decisions, applying moves by the player, and enforcing the game rules. (Note: this means that your view is not enforcing the game rules!)

All of these three components implement a relevant interface: IModel, IController and IView.

At the start of the program, one Model, one controller and one view are created. You can change which type of controller and which type of view is created (you will always use the same model). The other parts of the system treat these objects as the interface type (base class/super-class), NOT as the class which implements the interface. e.g., if you create a Reversi Controller, the model and view still see it as an IController interface, and can only functions which are on the IController interface. This means that you are (deliberately) limited in how you implement things – you have to work with the functions provided; you can freely switch between different implementations and all controllers should work with all views!

In the source code zip, you have been provided with:

- two example models. Both of these are very simple and will be sufficient. The test model creates some testing buttons for you. The normal model doesn't.
- two alternative views – one which is just a text-based entry to the console, and one is a text based entry within a GUI window. You will produce a third view to actually show the game more clearly as a GUI.
- one example controller – which will not enforce rules and will just allow pieces to be put anywhere. You will produce a second controller, which will enforce the rules of Reversi/Othello.
- Two test files – ones acts as a model+controller for testing your view class(es), and one as a model+view for testing the controller class(es).

When you have finished all of the requirements, go through the lines labelled **"Check:"** and check that each thing it tells you to check works correctly. This should ensure that you didn't miss something important.

When you think that you have finished your new view and/or controller, you can use the two test classes provided: TestYourController and TestYourGUIView to check for any obvious problems.

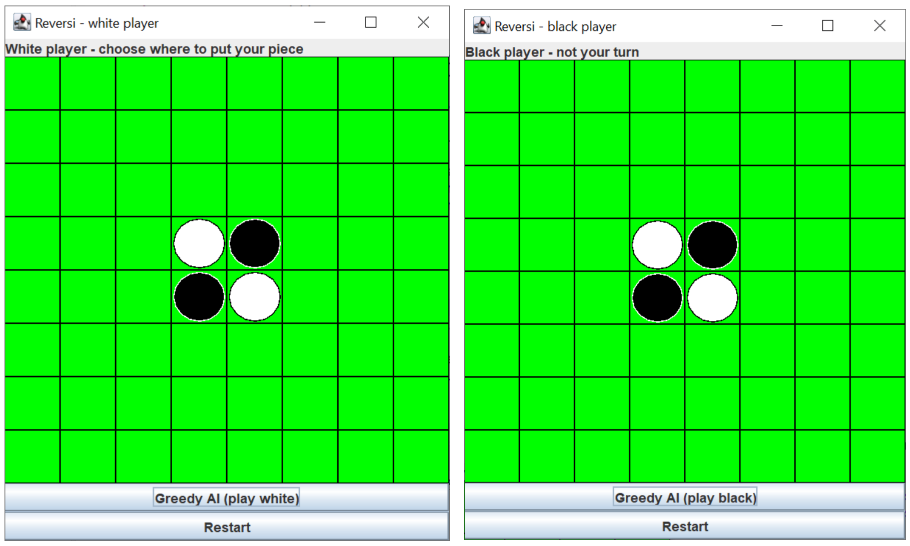
Also read the last page of this document before you submit! Check that your files for the controller work when the GUI files are not in the project, and vice versa for the GUI, because they will be tested separately, so you must not have dependencies between them.

Please watch the coursework intro lecture to see the rules of the game and how to use the varying controllers, models, and test classes.

General coursework requirements (overview):

You are going to produce a Reversi/Othello game by implementing a sub-class of IController (to enforce the rules of the game) and a subclass of IView (to show the view of the game). Your two classes will work together to create a full Reversi/Othello game. They must also work separately, combined with the IController and IView classes that I will use for testing.

1) The user interface will show two frames with different views of the board – one from the point of view of the white player and one from the point of view of the black player.



Each frame will have two buttons at the bottom, which will allow you to play as the AI and to restart the game. The restart button is right at the bottom, below the AI button.

A label across the top of each frame gives feedback to that player about what is happening.

At the start of the game you will have four pieces already on the board, in the positions shown in the diagram.

2) You may use **ONLY:**

- **standard swing/awt GUI classes,** standard Java classes **String, StringBuffer**, and **Random,** and **standard Java collection classes** (e.g. ArrayList, but I didn't need these, so you probably will not), and **System.out**/**System.err**.
- **the classes provided by me for your coursework** (in the zip file)
- **your own classes** (which may be subclasses of Swing classes)

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.

3) Don't try to access the file system or network, as doing so will cause your program to end when I test it. Classes which do this are not on the list of valid classes above anyway.

4) Only your main() function can be static. You must have no other static member functions! (Not doing this will penalise marks!) You MAY use static member variables if appropriate.

5) 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 also help you to understand the MVC pattern, but the main reason is so that I can test your submission.

6) 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.

7) You can run 'ReversiMain' to test your program. Just create the correct object types in it.

Program requirements:

You will implement the GUI and controller to allow you to play Othello/Reversi on the computer. 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 may develop the GUI and controller separately or together, it is up to you. There are advantages of either approach. Each of the following requirements is labelled according to whether it relates to GUI or controller. I suggest to first do requirement 1, creating both classes, then consider whether to do the two separately or together.

You could easily do the GUI first, then the controller, because the example controller supplied should be sufficient to test your GUI. However, building the controller before you build you GUI will be more complex – as the example GUIs are harder to use to test your controller.

At the end, double check that you met all requirements.

Notes on numbering:

> **Player numbers:** throughout this coursework, use the following values for player number: 0=none, 1=player 1, 2=player 2. Note that this is mentioned in the feedbackToUser() method of IView, various methods of IModel and various methods of IController. This gets used for current player number and for the contents of each board square (in IModel).
> **X and Y positions:** assume that all positions are 0-based, so that row and column numbers run from 0 to 7. You can assume that the board will be 8x8 for testing, so you only need to ensure that your code works for an 8x8 square board.

It should be possible/sensible to do the following requirements in order, but you can obviously do them in whatever order you wish:

1. **GUI and Controller:**
   - Unzip the contents of the supplied zip file into your src/source code directory (creating the reversi directory in your src directory)
   - Create your own GUIView class, in the reversi package, which implements IView and provide an empty implementation of each of the required methods.
   - Implement the initialise() method to store the parameters passed in attributes, which you will need to add to your class. See SimpleController for an example implementation.
   - Create your own ReversiController class, in the reversi package, which implements IController and provide at least an empty implementation of each of the required methods.
   - Implement the initialise() method to store the parameters passed in attributes, which you will need to add to your class. See Text for an example implementation, but do not copy the line "new Thread()…" from TextView, as that was only needed to handle the text.

Note that the objects to use and connect together are created in ReversiMain. i.e., ReversiMain specifies which objects will be used when it is executed, so you will need to change it to create your objects when you want to use them. In the main() function in ReversiMain you will create one of the controller, one of the model and one of the view objects, and then will connect them all together using the initialise function calls which follow. Please see the coursework intro lecture.

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 to create objects of your class. This is a good check of the naming – the correct name is already in ReversiMain, commented out.

Hint for getting an empty implementation of each method: implement the interface and use Eclipse's QuickFix to add the methods that are needed, then you can just fill in the details.

Looking ahead: By the end of this coursework, you will need to have a view class called GUIView which appropriately implements all methods of IView, to initialise the object (store the object references), refresh the appearance according to the data in the model, and set labels appropriately.
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.

Please read the comments before the functions in IView, IController and IModel to see what the various methods/functions should do and what you need to implement in your subclasses. This is particularly important for the IController's methods, and especially the update() method. Note also that ONLY the model should remember whose turn it is, and whether the game has finished. Any code in the controller or view that needs to know this should ask the model (using the object reference to the model, that was set in your initialise method, to call the relevant method on the model to ask it).

Check: all of your files are in a package called reversi (all lower case).
Check: your view class has the correct name of 'GUIView' and implements all three functions correctly.
Check: your controller class has the correct name of 'ReversiController' and implements all five functions correctly.
Check at the end: the 'update()' function works properly on your controller, so that if something is changed in the model then update() is called, your controller still works properly. Ideally this is easy, as long as your controller is not storing any information. Hint: using SimpleTestModel instead of SimpleModel for the model is an easy way to check this.
Check at the end: 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 (e.g. a class for a square on the board), but you need these two classes, with these names, so that I can test your submission, as I will access the methods of these classes to test your code.

Note: If you create a GUIView and use it, until you actually create a frame and show it, it won't do anything. So if you test your program now, it will probably end straight away – main() ends and because you didn't create a JFrame and show it yet, there is no other thread running. Note that TextView's initialise() method creates a new thread to listen for you typing something, which keeps that program running, and FakeTextView creates a JFrame and does a setVisible() on it (which keeps the program running as it waits to respond to the button press), so neither of those views have this problem.

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 of each frame – one for running a 'Greedy AI' and one to restart the game – clearing the board back to the first position.
   Hint: You will probably want to create these in the initialise() for the GUIView class, in the same way as I did in the FakeTextView class.
   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: You have 2 frames. Each has a label, a board and 2 buttons.

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.
   Hint: Remember that the model should be the class which remembers what is in each square of the board – so you may want your buttons to ask the model.
   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 – there are some advantages for you to use a button rather than a label as the super/base class though. For this I created a BoardSquareButton which started off as a copy of ColorLabel but extending JButton rather than JLabel, then I just added the extra functionality. Some students in the past used a JLabel subclass, but that makes some things harder later.
   You will need to draw a green square, a black border, and white/black circle with a border around the circle in the opposite colour. (I check for all of these!)
   <u>Draw the contents yourself</u> (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 is unnecessary so don't delay starting this. I just made my buttons their own listeners as it was a simple way to do it.

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, so worth trying if you can.)
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 the 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: the board state, who the current player ism 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 new contents according to what is stored in the model.

7. **Controller:** The controller will decide what to do if a player selects a square. This will be one of the following:

- If there is no valid location to play in, then change to the other player. Ideally this should never happen as the controller should switch to the other player anyway if there is no valid place to play - 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 a valid location to play, and the correct player, then play there. It is only a valid location to play if it can capture one or more of the opposing pieces. In this 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.
- If it is an invalid location but the correct player, then ignore the click and send a feedback message of "Invalid location to play a piece."

Check: Controller rejects the wrong player playing a piece – and sends feedback to player – see also point 16.
Check: Controller rejects playing a piece by current player in an invalid square and sends a feedback message saying 'Invalid location to play a piece.'
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 for counting and checking.

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), if the space is empty then calculate how many pieces will be taken if a piece is placed in that space, then choose as it's move 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!"
    - If a player tries to play in a square which is not valid (will not capture any pieces) while it is their turn, set the feedback message to: 'Invalid location to play a piece.'
    - 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 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 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.

Check: finally, test your GUI with the provided test class, and test your Controller with the provided test class.

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 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.

19. **Additional things to check that you did not do:** If you do any of the following, then you will lose some marks. You shouldn't need to do any of these, and if you do then it indicates that you either made an error, or you did something inefficiently.

    Ensure that you **did not**:

    a. Create more than two frames. You should only have two frames – one for each player. If you created more then you probably had something go wrong, or did not show them.
    b. Recreate the frames or controller when the game ends. You should instead be setting the variables back to the defaults.
    c. Storing information about the current player or board contents in the GUI or controller. This information should be in the model, and you should be showing that you know how to get the information from the model within your controller and/or GUI when it is needed. E.g., paintComponent() of the board square should be asking the model to tell if whether there is a piece there or not, rather than remembering it. Note: if you store info in the boards, then the options in SimpleTestModel won't all work properly, so you should be able to see the problems happen.
    d. Recreate the squares on playing a piece or finishing the game. You should be creating the squares of the board once, and reconfiguring them to show different things, not recreating new squares.
    e. Tried to use images rather than drawing the squares and circles. If you did this then you didn't get the required practice in Java that you were supposed to get. Also, it probably won't work in the secure test environment that I will run it in. Only Java code files get copied over to the test environment, not any images that you may include.
    f. You didn't put the border around the oval for the piece. Doing this shows that you understand how to do this, so not doing it means fewer marks.

    You may still decide to do some of these, if they makes it easier for you, but be aware that it will lose you marks as well as probably failing some of the tests as well.

Additional Hints for the controller:

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.

Final tests before submission:

You should test your controller using the supplied "TestYourController" class, as shown in the coursework intro lecture. This will do various tests to warn you about obvious problems, but is in no way complete – the testing of your submission will be must more thorough.

You should also test your GUI view using the "TestYourGUIView" class, as shown in the coursework intro lecture. This just gives you a few feedback messages so that you can check that things work as expected, without involving your own controller.

Before submission you should also ensure that your GUIView and Controller are separate.
1) Create a project with only the files I initially provided, and the GUI classes that you created. Test that everything runs properly (without your ReversiController).
2) Create a project with only the files I initially provided, and the ReversiController classes that you created. Test that everything runs properly (without your GUI classes).
In the past, students accidentally introduced dependencies with their other classes, which caused it to fail to compile and/or run when I tested it. doing this will ensure that you didn't do so.