Exercise 1

1. The objectives for this test are primarily to test the functionality, boundaries, usability and performance of my program. This involves verifying that every functionality of the chess game is working properly, like ensuring that player can only make valid moves and checkmate and stalemate raise game ending flags. We also want to test for different boundaries by putting the program in different scenarios to ensure that it can handle all of them efficiently and correctly. Regarding performance, we want to assess its responsiveness and complexity. And finally, we also want to ensure how intuitive and user-friendly the interface of the game is.

2. The test will take place by following procedures from a combination of software testing types. These include unit testing: to verify that each class or unit of code behaves as expected, independent of the rest of the program; Integration testing: to verify that classes are interacting with each other in the desired way; Release testing: to identify problems with user interface; And performance testing, specifically performance tuning: to optimize the program by identifying and resolving inefficiencies, like algorithmic complexity. These tests will take place will the code is being developed.

   The classes chosen for this testing are the Board class, as it handles most of the game logic behavior, the CheckHandler class (to test that check, checkmate and stalemate scenarios are always handled properly) and the MouseHandler class (to test the boundaries of the mouse behaviour).

3. The tools that will be used are primarily built in Java functions, like “System.nanoTime()”, to get an idea of how many nanoseconds a process takes for completion. Other than this, most of the test analysis will be done by putting the program into different scenarios via modification of the code.

4. The tests will be declared successful or failures if the actual results coincide with the expected ones. If they are the same, it will mean that the behavior of the program aligns with the requirements set in the “SRD” document. Furthermore, it is important to note that the failure of a test does not indicate a critical failure of the entire system. It indicates a specific issue within the scope of the test case, unless proven otherwise.

Exercise 2

Test Case 1

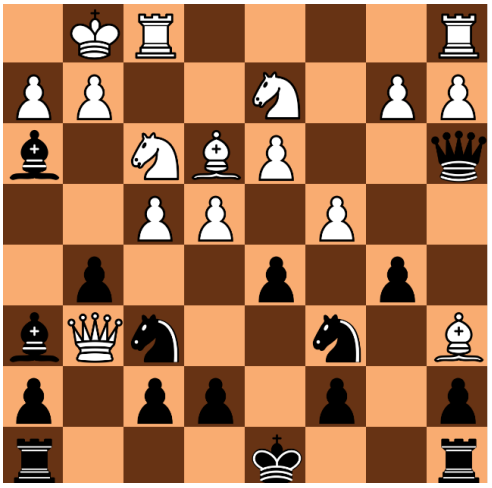
Test title: Drawing of the pieces in the correct places.

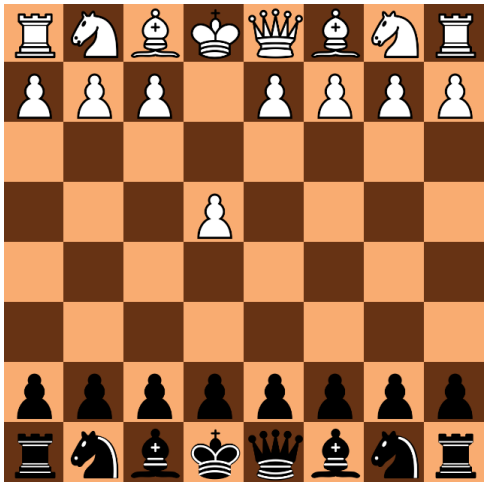
Test Summary: The unit tested is the Board class, specifically the draw method. The board will undergo various pieces configurations to ensure that the pieces are displayed in their corresponding position every time.

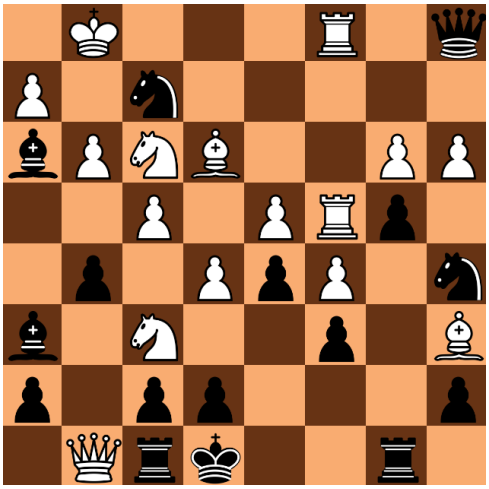
Test Steps:

- Change the value of rows and columns from different pieces.
- Run the program.
- Check whether the game panel has displayed the new pieces positions in the appropriate tiles.
- Go back to step 1.

Test Data:







Because there are around 10^111 different configurations for a chess board, not all of them can be tested because of computing power and time limitations. However, the images above show some of the examples that were tested for. This test appears to be successful for every case, meaning that there is no problem with the draw method of our board class.

Test Case 2

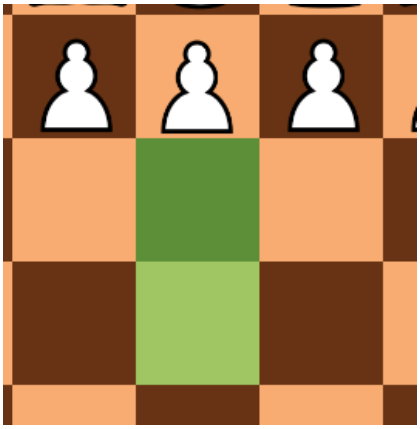
Test title: Valid move checking.

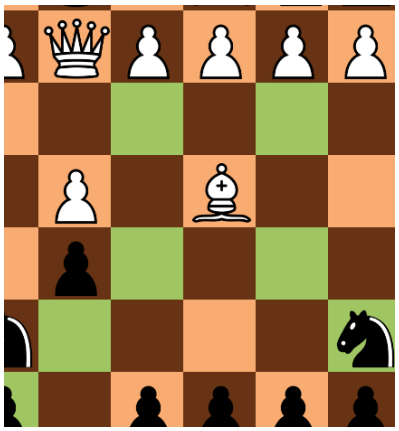
Test Summary: The unit tested is the Board class, specifically the isValidMove method. The board will undergo various attempts of move validation to ensure that every rule of chess is followed when deciding if a move is valid or not.

Test Steps:

- Set up a specific scenario to test a move.
- Run the program.
- Click a piece.
- Iterate through every tile of the board and call the isValidMove method for that tile.
- Paint the tile tested green if the method returns true.
- Check whether every move that should be legal is painted green, and every move that should be illegal is not painted.
- Go back to step 1.

Test Data:





The images above are some examples of the test that was carried out. After a vast number of trials, the isValidMove appears to return correct outputs for every tile being tested. This will also mean that it is well integrated with the Piece class, as methods for each specific piece are called inside the move validation function.

Test Case 3

Test title: Making a move.

Test Summary: The unit tested is the Board class, specifically the makeMove method. The board will undergo various attempts of move creations to ensure that pieces' positions are correctly modified after a move, and that if a move involves taking an enemy piece, that the enemy piece is erased from the board.

Test Steps:

- Set up a specific scenario to test a move.
- Run the program.
- Click drag and release a piece in a valid tile.
- Check whether the piece is displayed correctly in its new position, and if the move implied taking a piece, check whether the taken piece is erased from the board.
- Go back to step 1.

Test Data:

The test appears to be successful. Every tested move has shown to behave in the expected manner. Whenever a piece is taken, this object is set to null and erased from the board thanks to this line of code:

```
pieces[row][col] = pieces[piece.row][piece.col];
pieces[piece.row][piece.col] = null;
```

Test Case 4

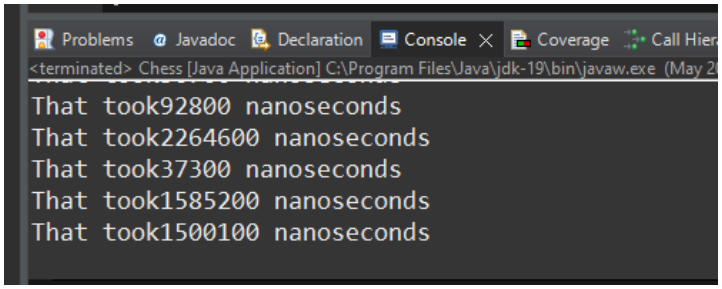
Test title: Performance of making a move.

Test Summary: We want to test how many nanoseconds does it take to make a move after the function is called. We want to keep this number under 1 000 000 (1 millisecond) to ensure fast responsiveness from the program.

Test Steps:

- Make a move.
- Declare a startTime variable by getting the current time of the program in nanoseconds at the beginning of the makeMove method.
- Declare an endTime variable by getting the current time of the program in nanoseconds at the end of the makeMove method.
- Substract the time difference and display it in the console.
- Check whether the time varies depending on the move made.
- Go back to step 1.

Test Data:



As shown from the image above, some of the moves are taking 1 or up to 2 milliseconds. A trend was discovered regarding the amount of time taken, and the type of move. If the move implies taking a piece, the method takes more time to complete.

Time	Piece taken?
46400	0
77800	0
2656200	1
50700	0
1798300	1
50400	0
105500	0
41600	0
2759500	1
1599900	1
1636000	1
80500	0
1650400	1
1445700	1

SUMMARY OUTPUT

Regression Statistics	
Multiple R	0.935588262
R Square	0.875325397
Adjusted R Square	0.864935846
Standard Error	381234.3961
Observations	14

ANOVA

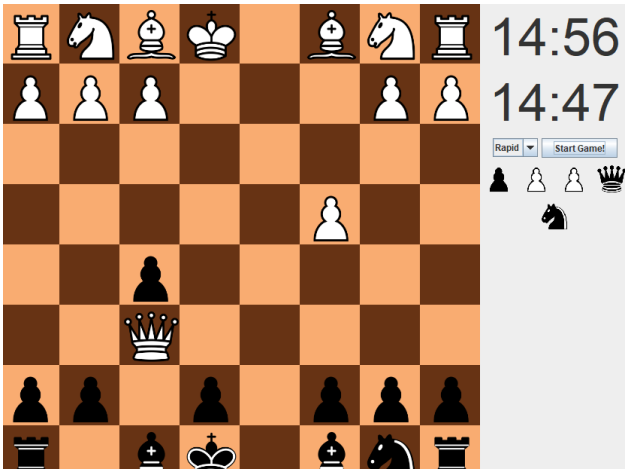
	df	*SS*	*MS*	*F*	*Significance F*
Regression	1	1.22449E+13	1.22449E+13	84.25055684	8.96538E-07
Residual	12	1.74408E+12	1.4534E+11		
Total	13	1.3989E+13			

	Coefficients	*Standard Error*	*t Stat*	*P-value*	*Lower 95%*	*Upper 95%*	*Lower 95.0%*	*Upper 95.0%*
Intercept	64700	144093.0576	0.449015387	0.661421943	-249251.8026	378651.8026	-249251.8026	378651.8026
Piece taken?	1870442.857	203778.3563	9.178810208	8.96538E-07	1426447.96	2314437.754	1426447.96	2314437.754

A regression analysis was performed to validate our hypothesis. In this case, because the P value is greatly lower than 0.05, we can in fact assume that taking a piece has a great influence on the time taken. Now, let's look at the code and see why. If we take a close look at the method, the if statement that determines whether a piece is being taken is:

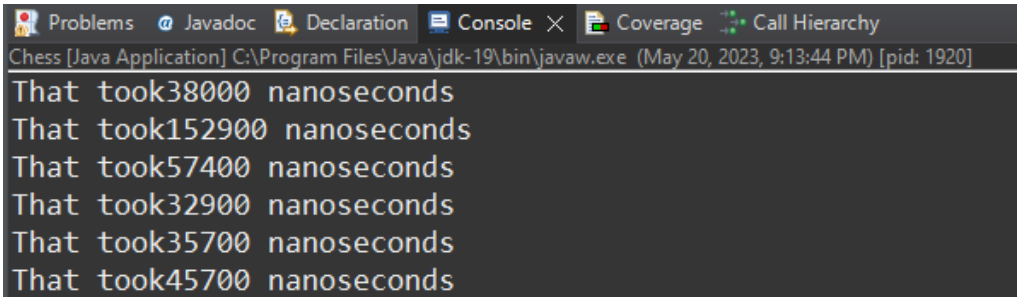
```
if(pieces[row][col] != null) {
    if(pieces[row][col].isWhite()) {
        takenWhitePieces.add(pieces[row][col]);
        uipanel.addTakenPieces(this, takenWhitePieces.size()-1, 1);
    }else {
        takenBlackPieces.add(pieces[row][col]);
        uipanel.addTakenPieces(this, takenBlackPieces.size()-1, 2);
    }
}
```

This will be the only thing that distinguishes a regular move from a move that involves taking a piece. Now let's identify what is this piece of code doing. The "takenWhitePieces.add(pieces[row][col])" is simply adding an element to a list we have defined for taken pieces. As we know, the time complexity for adding an element is in O(1), so this is not a problem. Now, the next line might be what is causing the performance to be affected. The size function, in java, is in O(1) too, as a "size" variable keeps track of the size of the list at all times. However, when we call the uipanel method, we are adding a new label to the panel (containing an image of the taken piece), just like this:



This process is what's causing this delay. This can be proven by commenting on the add label call, as we can observe vast improvements in the performance. However, we don't want to get rid of this functionality, as displaying the taken pieces is part of our requirements. So, we need to find another way of reducing this delay.

A new class was introduced, "PiecesPanel", this class will display the taken pieces via the paint function inherited from the "java.awt.Graphics" package. This way, every time a piece is taken, we will just call the "repaint" method, which is much faster than adding a new label. Now, after these new changes:



As one can observe, the "makeMove" method is consistently taking under 1 millisecond, which aligns with our test benchmark.

Test Case 5

Test title: Mouse handling boundary testing.

Test Summary: We want to test the boundaries of the MouseHandler class. This involves clicking tiles where there are no pieces, clicking pieces of a color that is not in turn, dragging the mouse outside the frame, and releasing the mouse on illegal tiles. The test will be successful if we can't find any abnormal behavior.

Test Steps:

- Click illegal tiles.
- Check different scenarios of mouse dragging.
- Release the mouse in illegal tiles and out of the boundaries of the frame.
- Check whether for abnormal behavior.
- Go back to step 1.

Test Data:

After rigorous testing, everything appears to work properly, except for an out of bound error when releasing the mouse outside the frame. To solve this, the following defensive lines of code where added:

```
if(e.getX() < 0 || e.getX() > 640 || e.getY() < 0 || e.getY() > 640) {

    board.pieceToMove.x = board.pieceToMove.row * board.getTilesize();
    board.pieceToMove.y = board.pieceToMove.col * board.getTilesize();
    board.pieceToMove = null;

    return;
}
```

This will check for out of bounds position in the mouseReleased method.

Test Case 6

Test title: King in check method

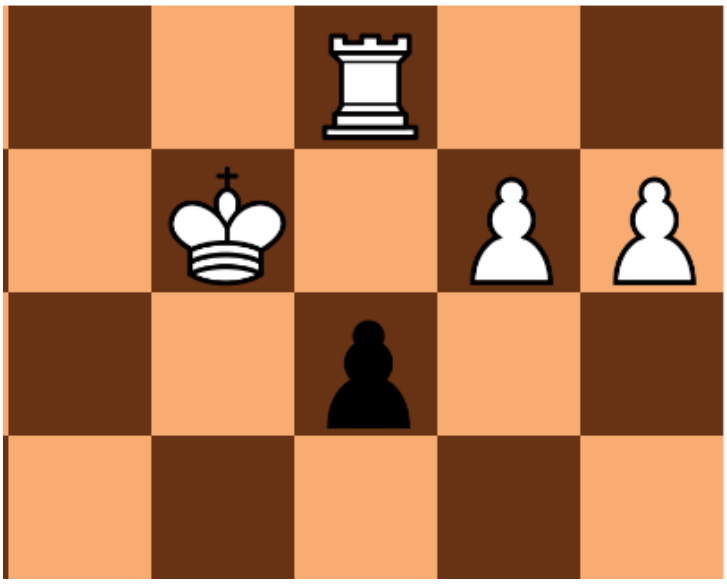
Test Summary: Our unit will be the CheckHandler class. We want to test the functionality of the kingInCheck method, to see if it returns an appropriate value for different scenarios.

Test Steps:

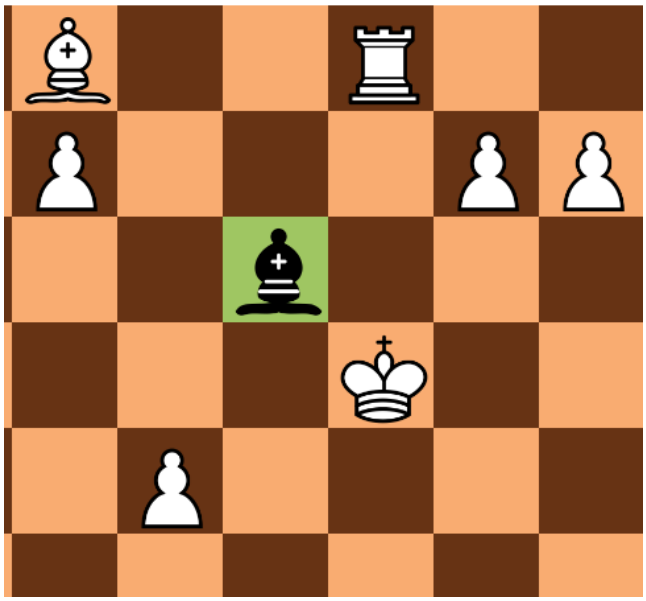
- Click a piece.
- Check if moves that will leave the king in check are valid.
- Go back to step 1.

Test Data:

After testing a vast number of scenarios, it appears that we can't make moves that leave the king in check. However, an error related to the pawns was found. To have a better understanding, let's look at this case:



As one can observe, the rook should be able to take the pawn in this scenario, as it will leave the king out of check, we can ignore the rest of the board for now. However, the method was still returning true for that tile, meaning that it wasn't recognizing that the pawn would be taken after the move.



As one can see from this other image, this wasn't the case for any other piece, like this bishop for example. The white bishop can take the black, as the program knows the king won't be in check anymore. This was fixed by adding a new logical condition to the method:

```
&& !(row == i && col == j))
```

Which basically means that, if the move's row and column is equal to the row and column of the pawn, we don't need to return true. After this change, we can see that the problem was fixed: