System analysis using UML diagrams and CRC

# Design Pattern: < Strategy Pattern >

1. *Provide a brief description of your pattern*

It is used to define a family of algorithms, encapsulate each one, and makes them interchangeable.

1. *Explain why is important for your chess game*

It is important because it can let the user change the algorithm based on the behavior (can be more flexible and reusable) in the chess board, which is sometimes useful to do so.

1. *Explain how did you have to modify your code to adapt the pattern*

I created an interface called Piece (Strategy) from which it defines the movements for the pieces and I implemented all the chess pieces classes to it (that I already had before). After that, all the chess piece classes (sub-classes) have their own move methods and will print out the result (when the program is run and the piece and color is picked by the user, which can be interchangeable). Then, the interface class needs the information that is passed from my chessboard class. For that, all I did was setup a new piece based on its type for each one with its (color, initial x, initial y), then I would call that specific piece method based on its type and color with its (final x, final y) position.

1. *Provide the general UML diagram of the pattern listing each of the components. Do not include any chess classes. You will show it latter.*

|  |
| --- |
| <<Piece>> |
|  |
| whitePieceMove(finalcolumn, finalrow)  blackPieceMove(finalcolumn, finalrow) |

# UML diagram of your system

# 

Include three different types of java collections

Collection 1 < ArrayList>

1. I selected this pattern because < I want to store a collection of piece elements >
2. I used < for each > to traverse my collection because < It is shorter and easier to understand >

Collection 2 < LinkedList >

1. I selected this pattern because < I want to store a collection of x and y elements >
2. I used < iterator> to traverse my collection because < I want to use the built-in methods that it has >

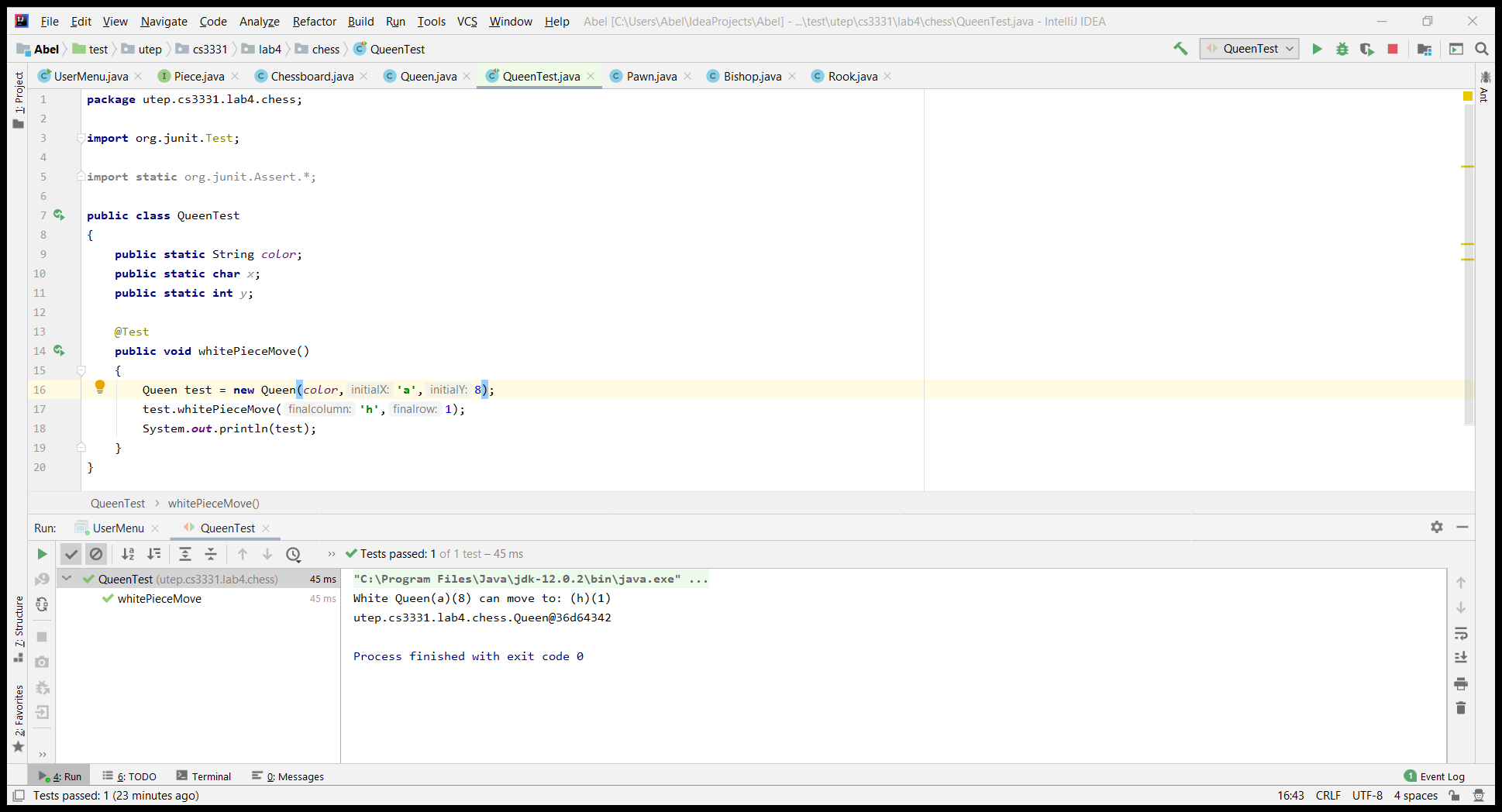
Collection 3 < HashMap >

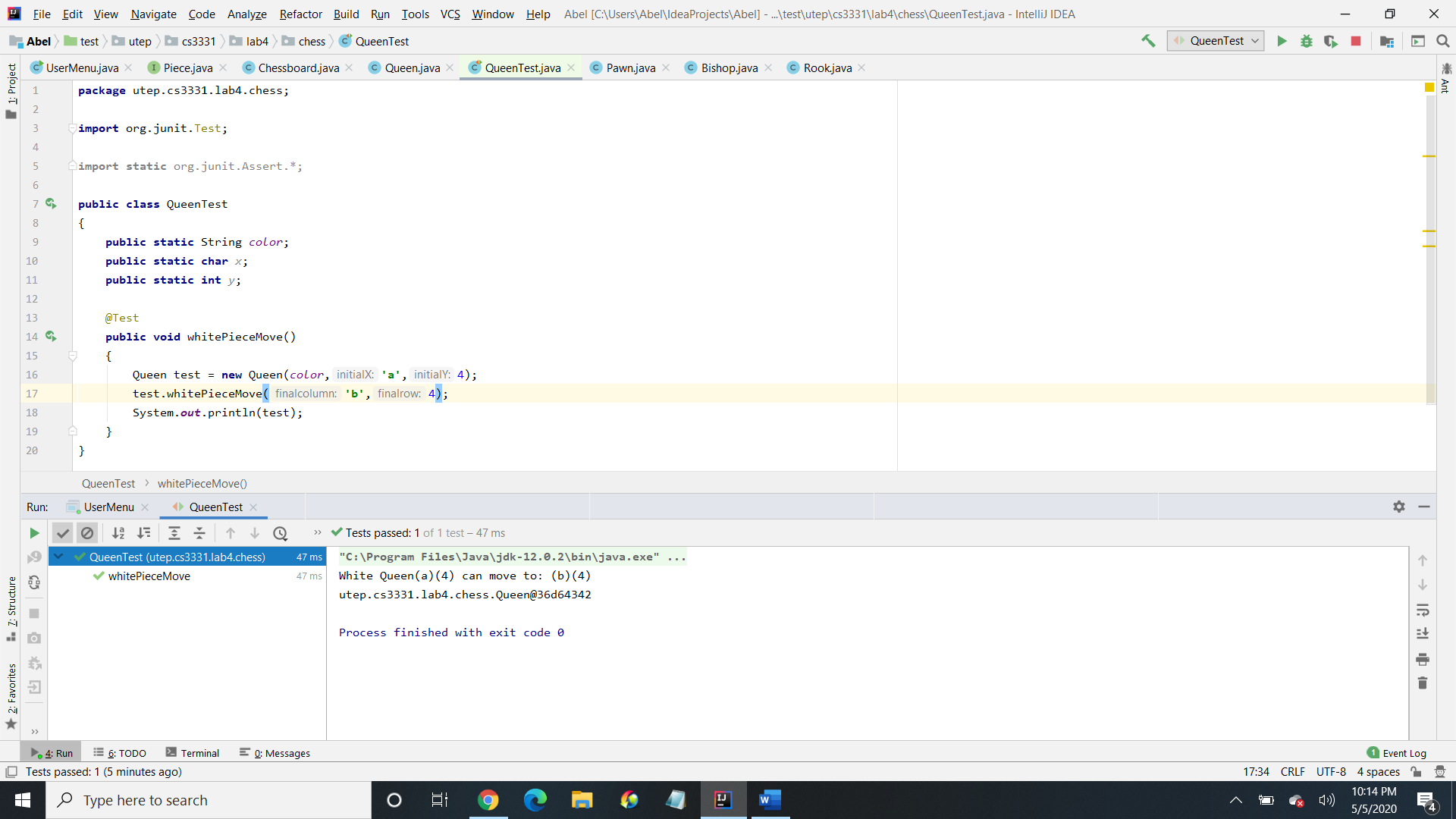
1. I selected this pattern because < I want to store keys and values into a list of maps >
2. I used < aggregation operator > to traverse my collection because <I want to iterate every element on the stream>

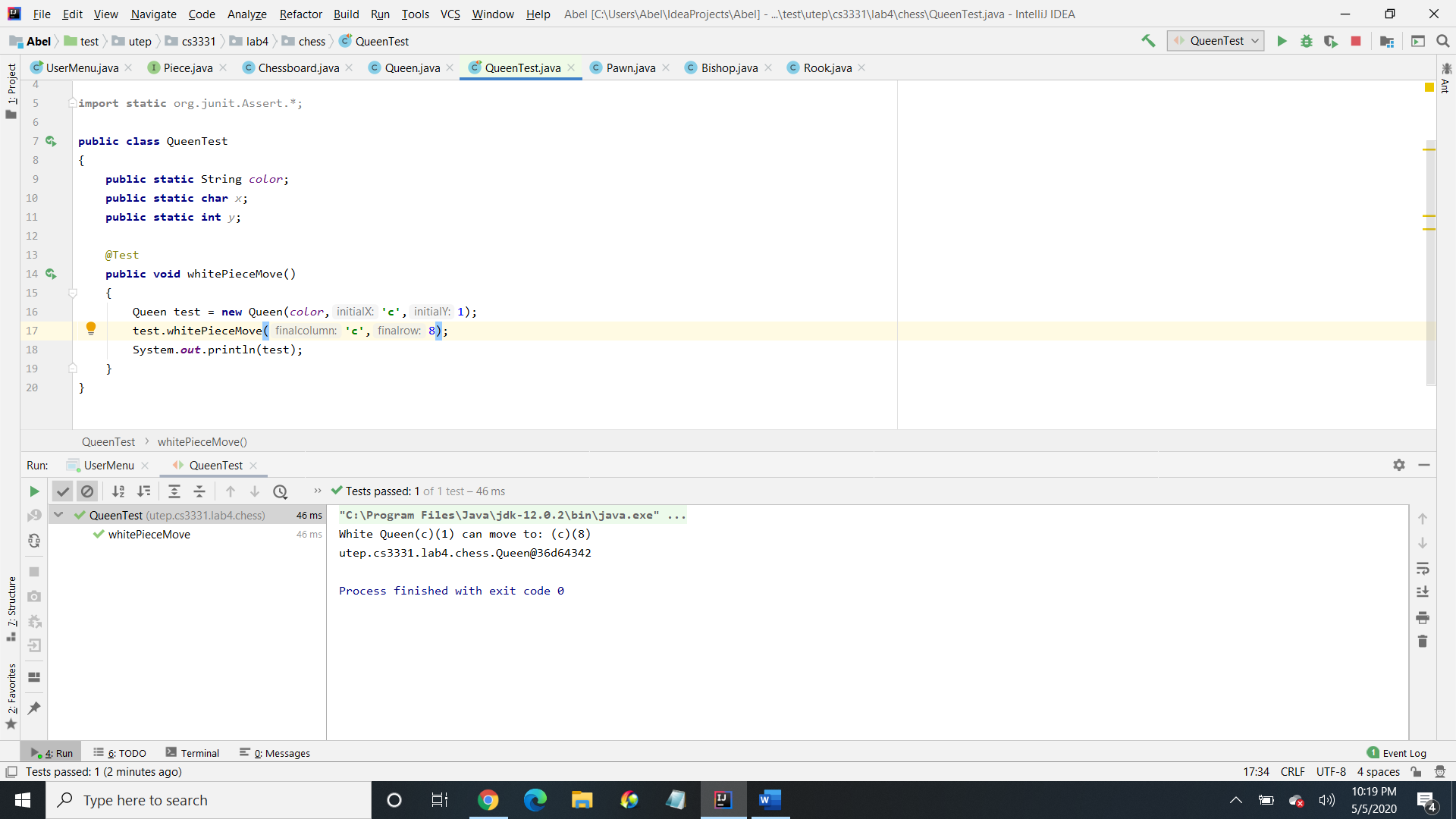
# Use Junit to test three of your most complex methods –

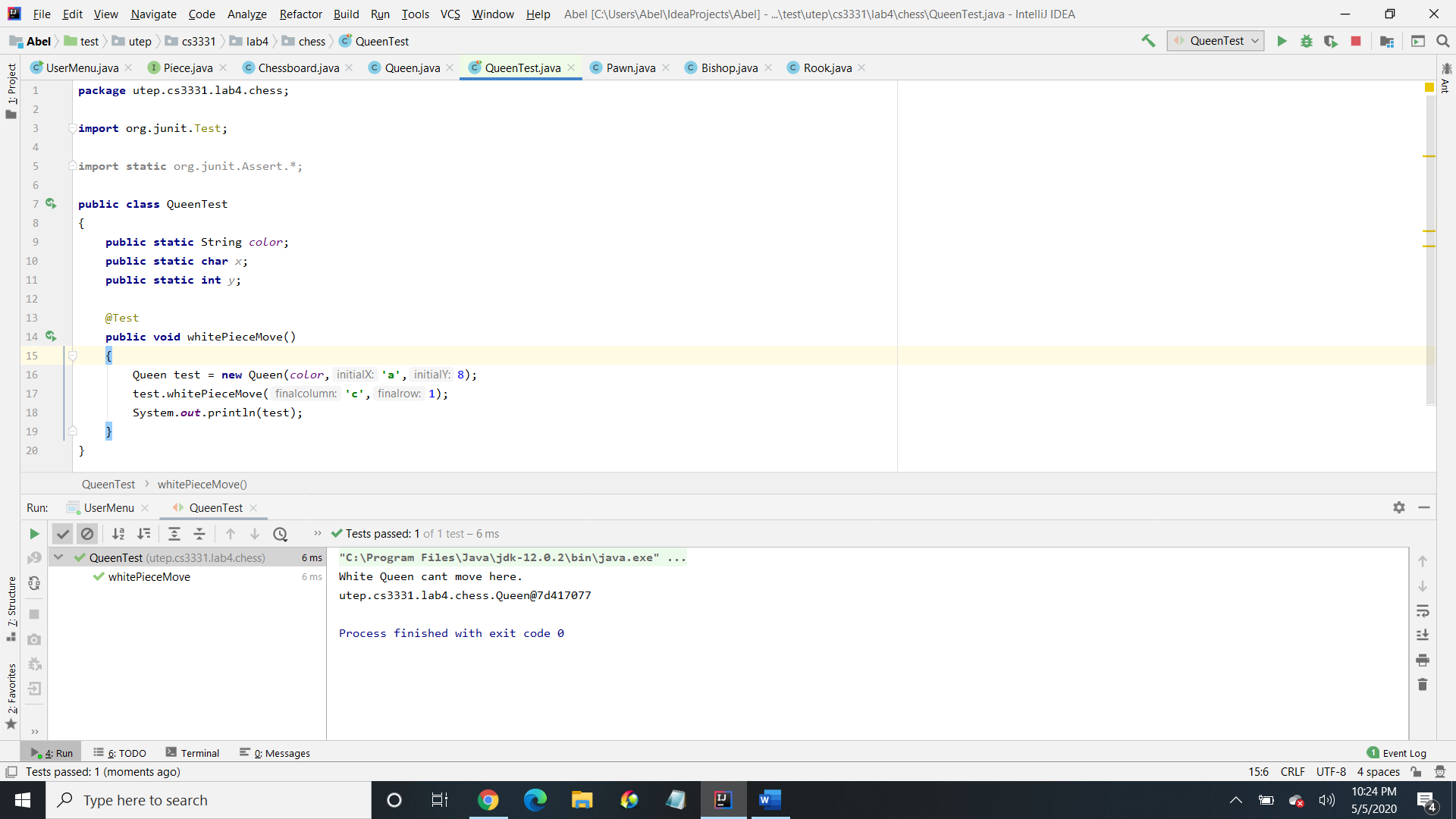
***1. Method name <whitePieceMove >***

*I selected this method because < The Queen has all the movements >*

*Test case 1 <QueenTest >   
< My strategy is to move the white queen at (a)(8) to (h)(1) which I used a for loop to move down to the right >  
< I used this strategy because the for loop iterates the movement till its correct (incrementing), depending on the initial and final positions of x and y >  
< >*

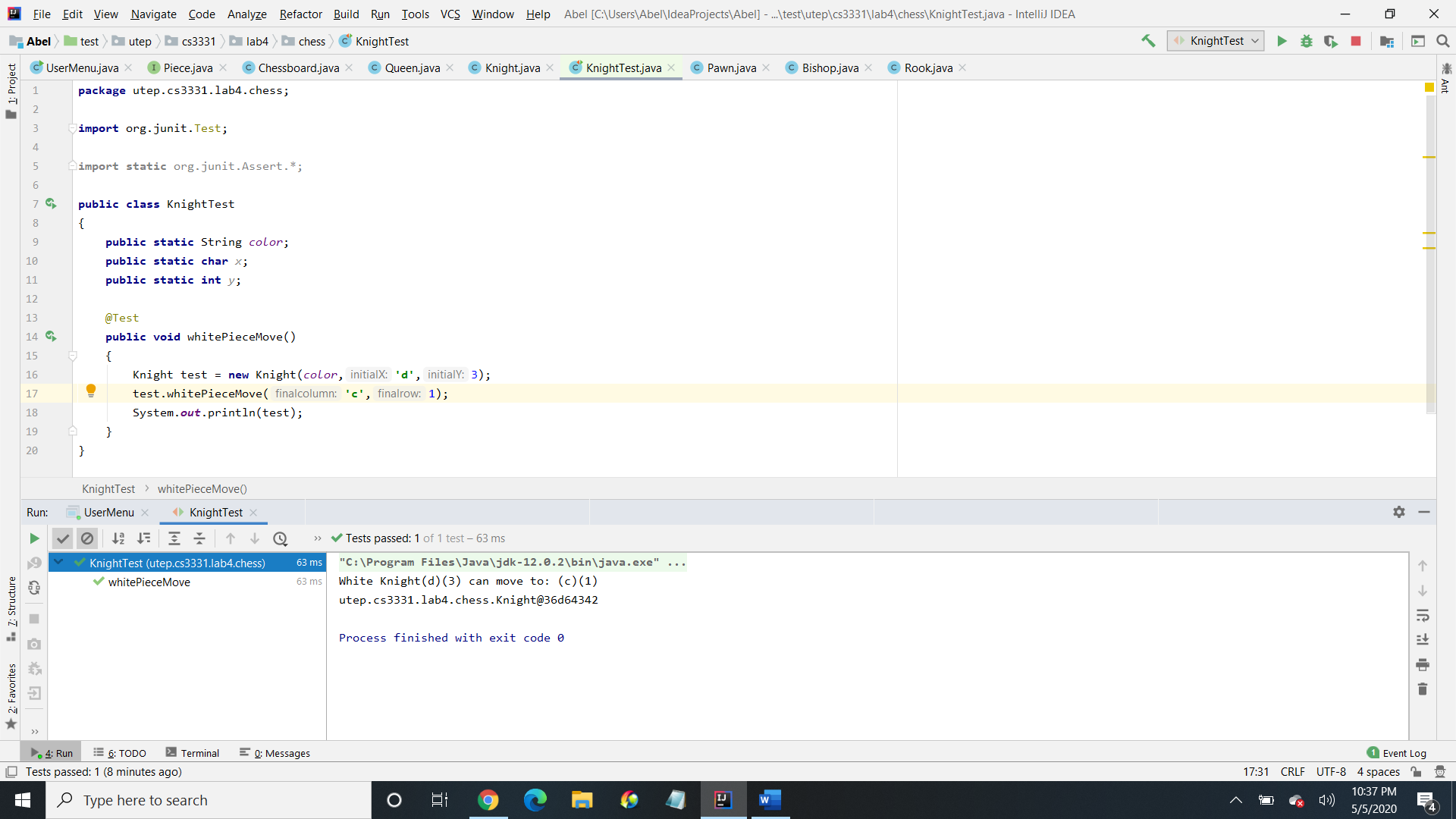
*Test case 2 <QueenTest>   
< My strategy is to move the white queen at (a)(4) to (b)(4) which I used an if statement to move horizontally to the right >  
< I used this strategy because the if statement checks if x is equal to final column or y is equal to final row (which in this case it is) >  
< >*

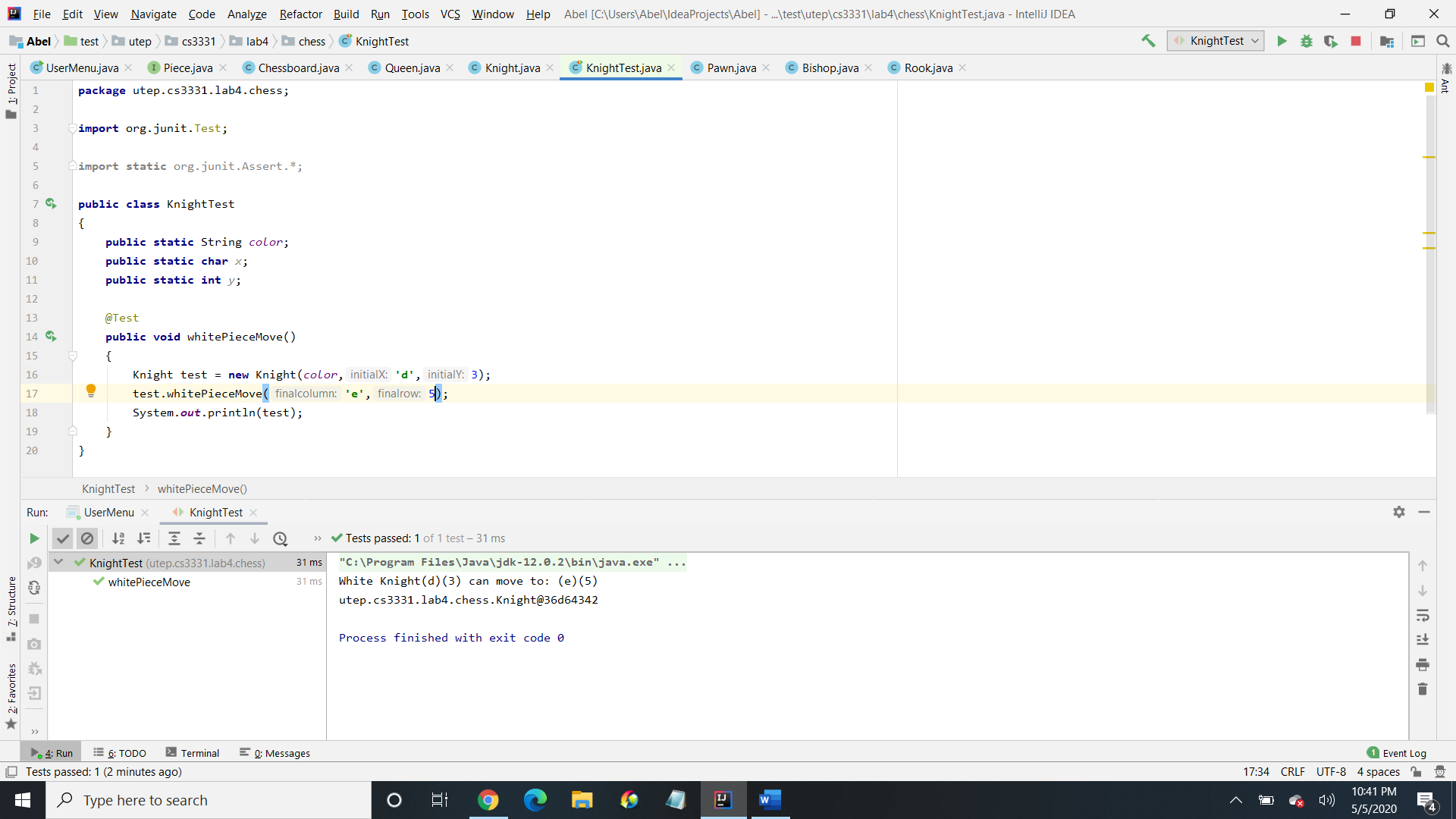
*Test case 3 <QueenTest >   
< My strategy is to move the white queen at (c)(1) to (c)(8) which I used an if statement to move up vertically >  
< I used this strategy because the if statement checks if x is equal to final column or y is equal to final row (which in this case it is) >  
< >*

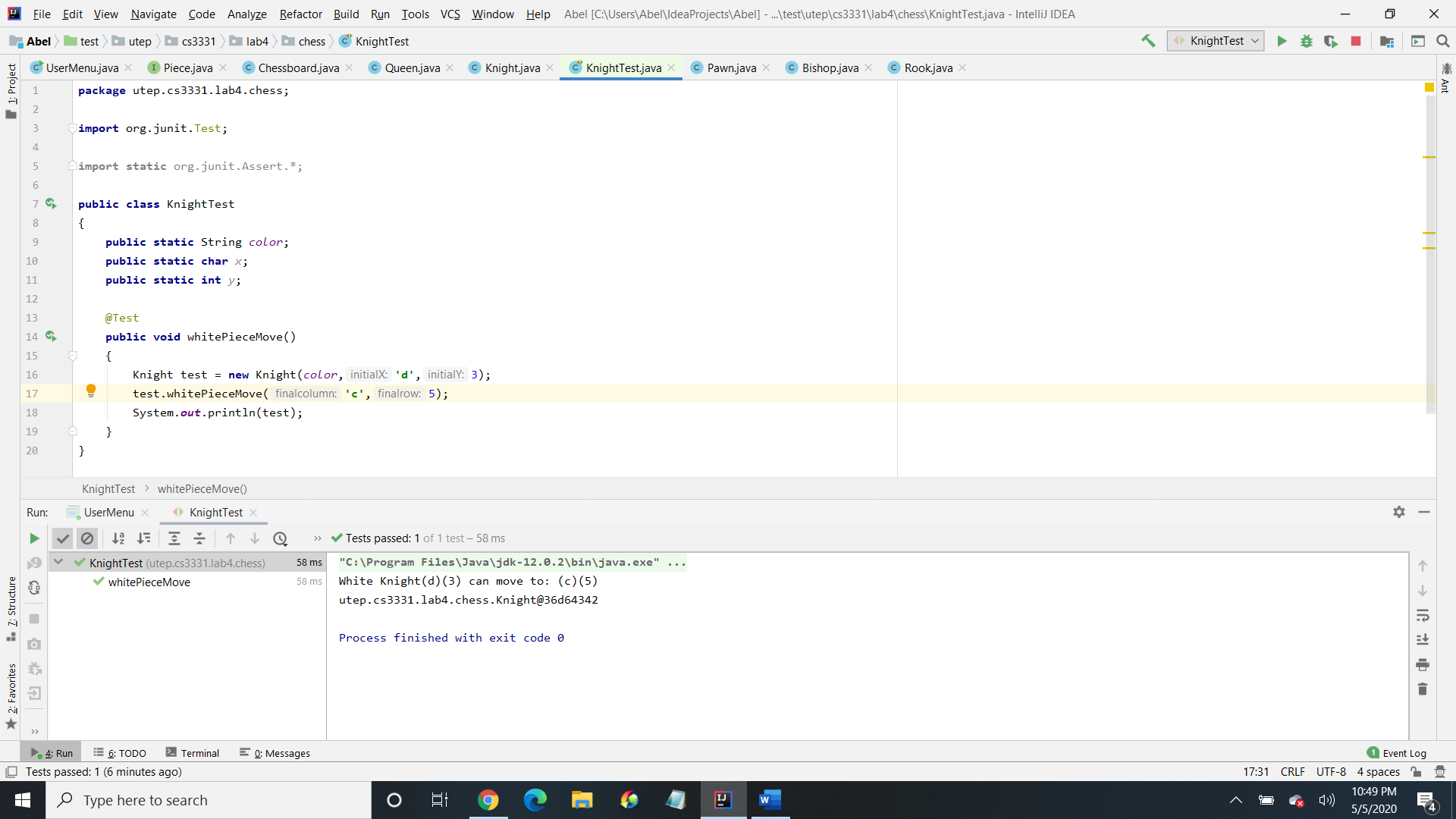
*Test case 4 <QueenTest >   
< My strategy is to move the white queen at (a)(8) to (c)(1) which in this case the white queen cant move here >  
< I used this strategy because I want to see if the white queen can move to that position which it cannot >  
< >*

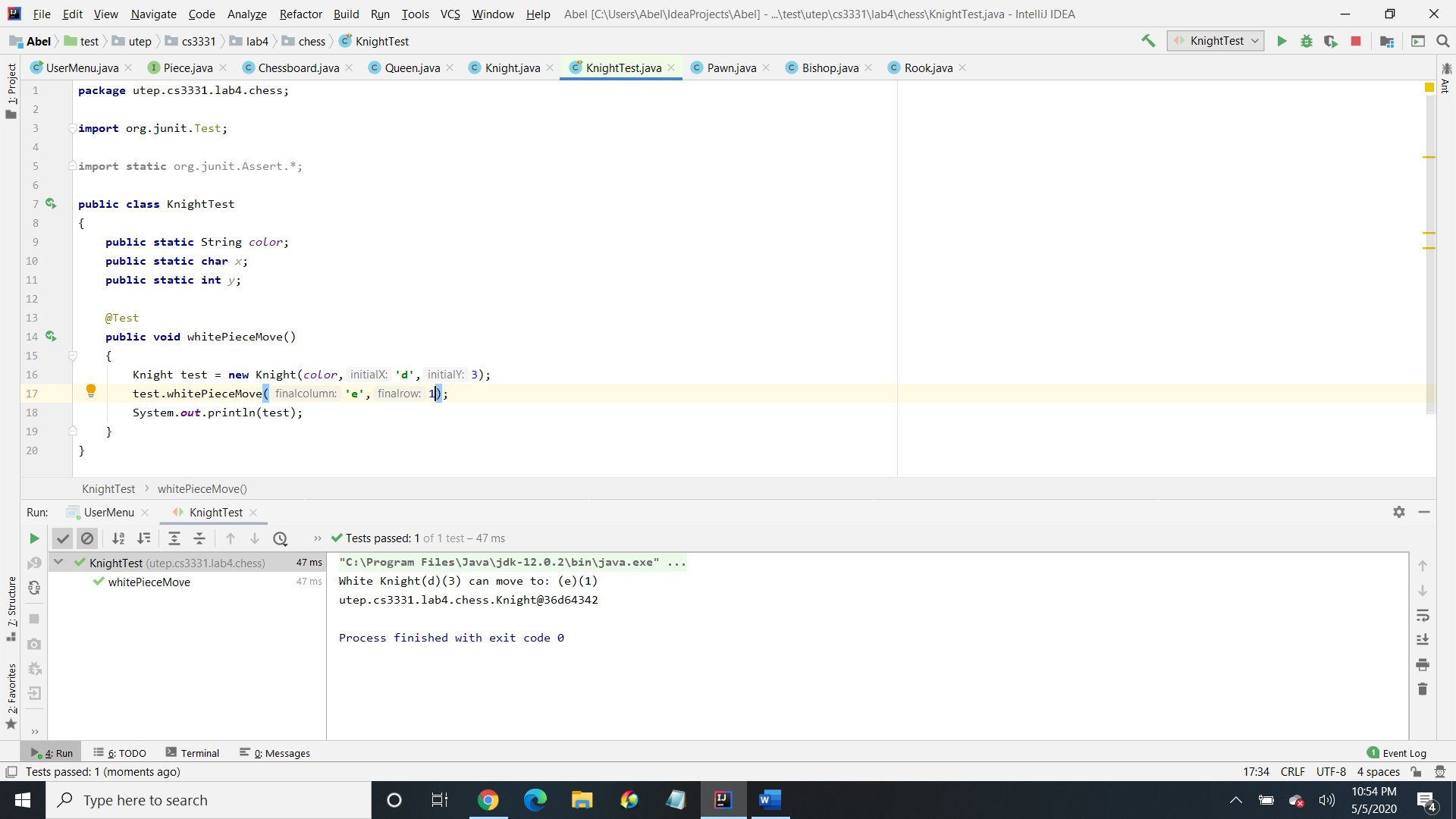
***2. Method name <whitePieceMove>***

*I selected this method because < The knight has different movements >*

*Test case 1 <KnightTest >   
<* *My strategy is to move the white knight at (d)(3) to (c)(1) which I used a for loop to move down to the left >  
<* *I used this strategy because the for loop iterates the movement till its correct (incrementing), but this time I am subtracting initial (x) -1, initial (y) – 1, final column -1, and final row – 2 (which is the knights movement) depending on the initial and final positions of x and y >  
< >*

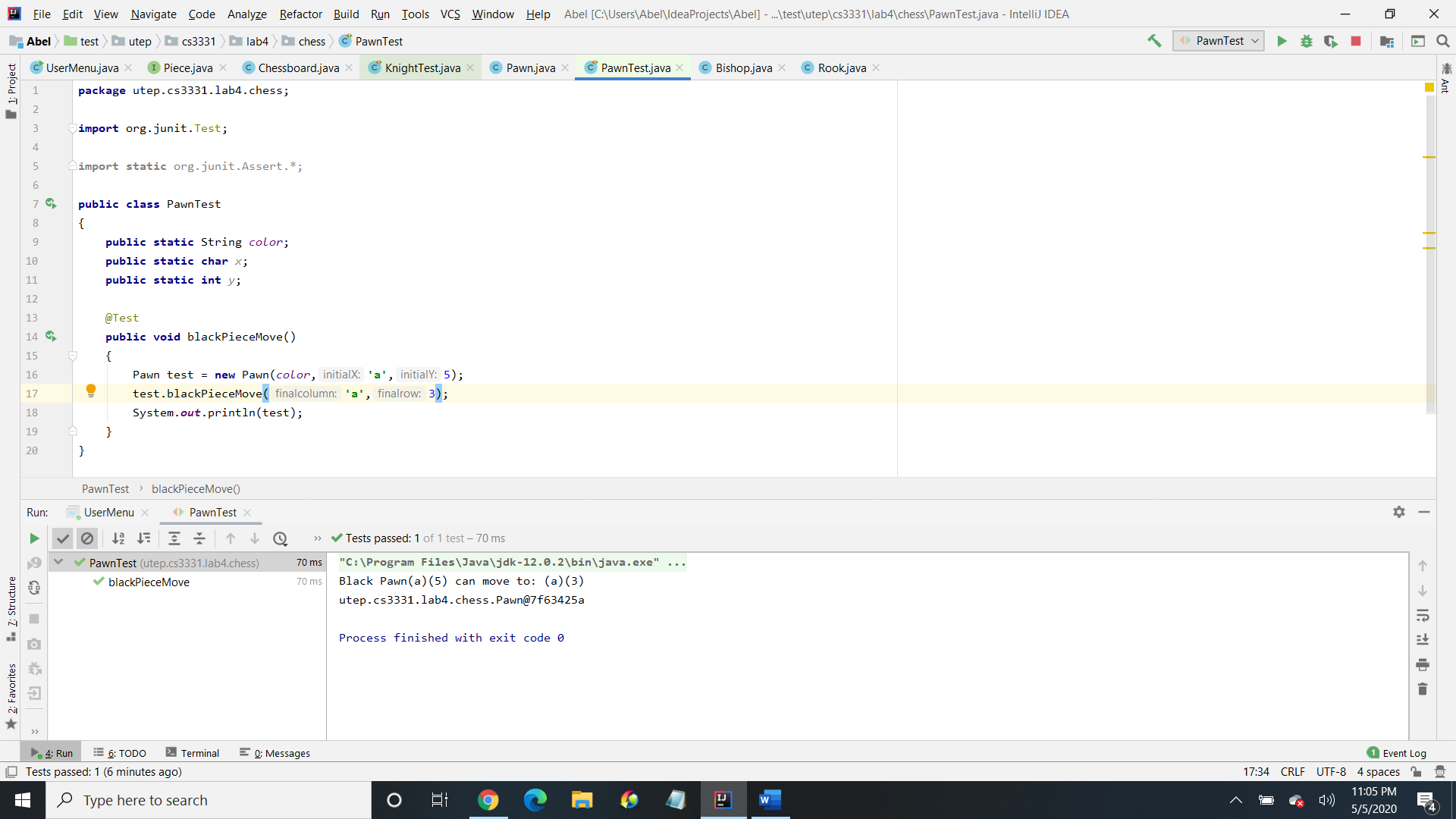
*Test case 2 <KnightTest>   
<* *My strategy is to move the white knight at (d)(3) to (e)(5) which I used a for loop to move up to the right >  
< I used this strategy because the for loop iterates the movement till its correct (incrementing), but this time I am adding initial (x) +1, initial (y) + 1, final column +1, and final row + 2 (which is the knights movement) depending on the initial and final positions of x and y >  
< >*

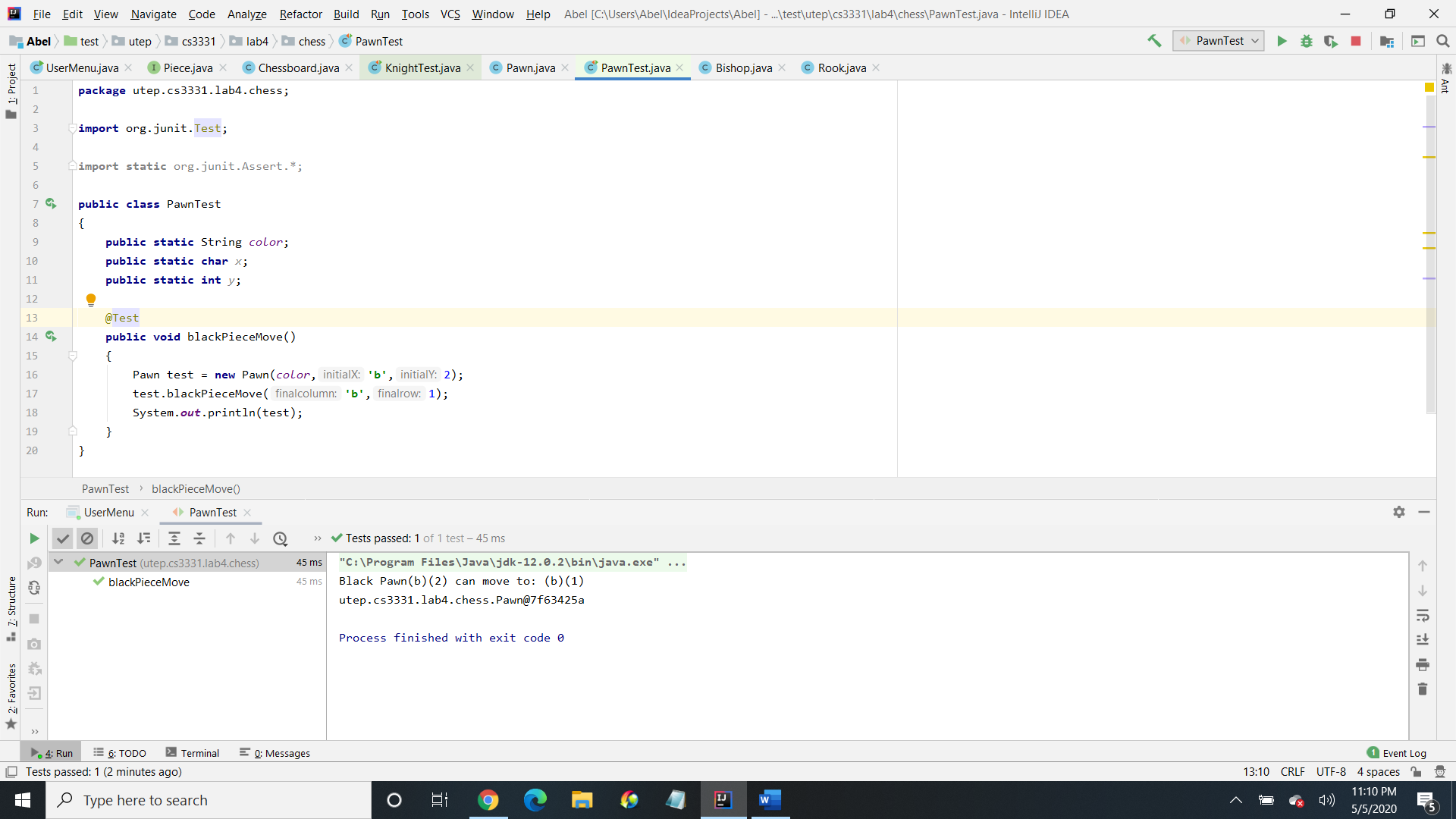
*Test case 3 <KnightTest >   
< My strategy is to move the white knight at (d)(3) to (c)(5) which I used a for loop to move up to the left >  
< I used this strategy because the for loop iterates the movement till its correct (incrementing), but this time I am subtracting initial(x) -1, then adding initial(y) +1, then I am subtracting final column -1 and adding final row + 2 (which is the knights movement) depending on the initial and final positions of x and y >  
< >*

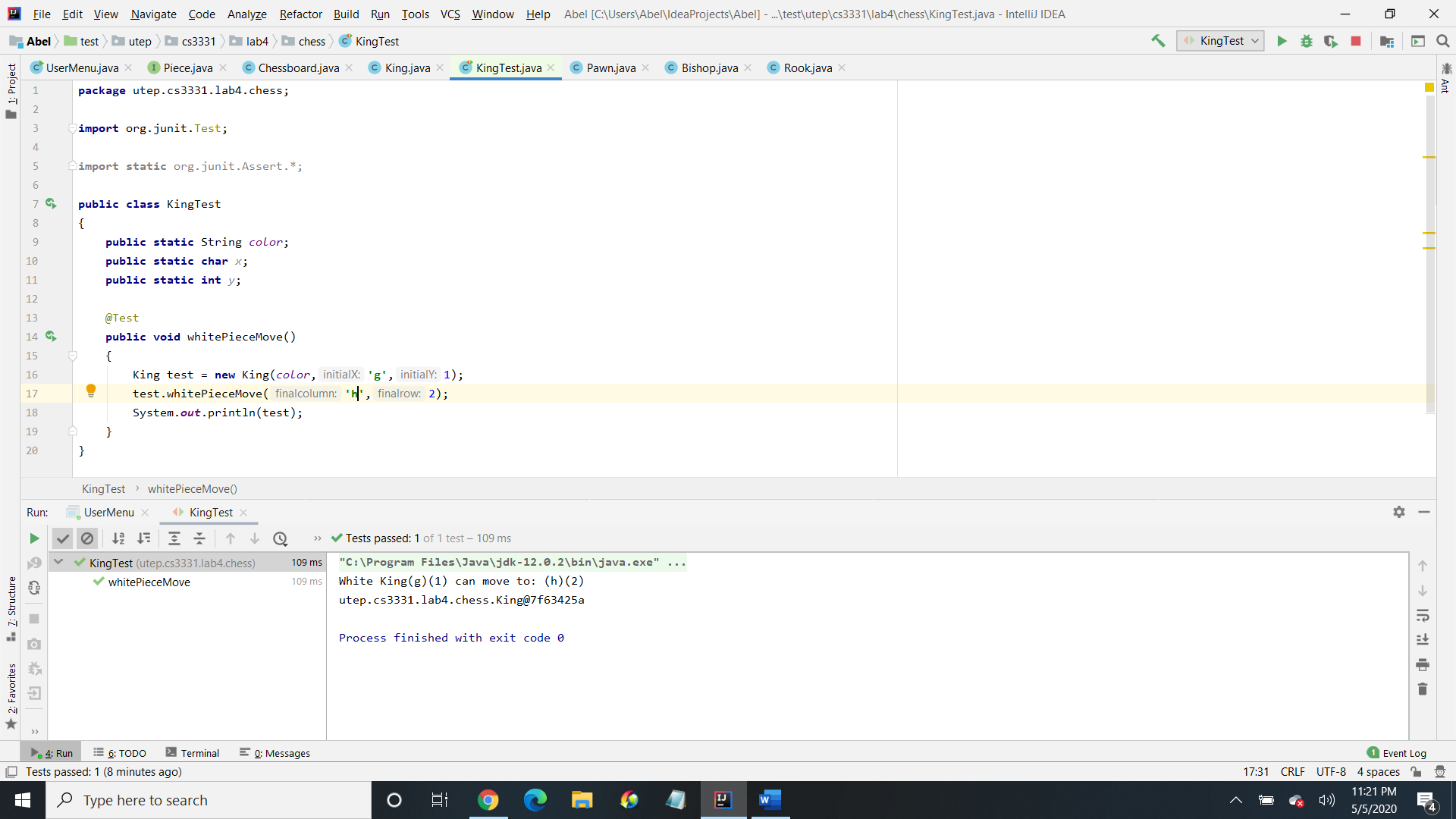
*Test case 4 <Knight Test >   
< My strategy is to move the white knight at (d)(3) to (e)(1) which I used a for loop to move down to the right >  
< I used this strategy because the for loop iterates the movement till its correct (incrementing), but this time I am adding initial (x) +1, then subtracting initial (y) – 1, then adding final column +1, and subtracting final row – 2 (which is the knights movement) depending on the initial and final positions of x and y >>  
< >*

***3. Method name < blackPieceMove >***

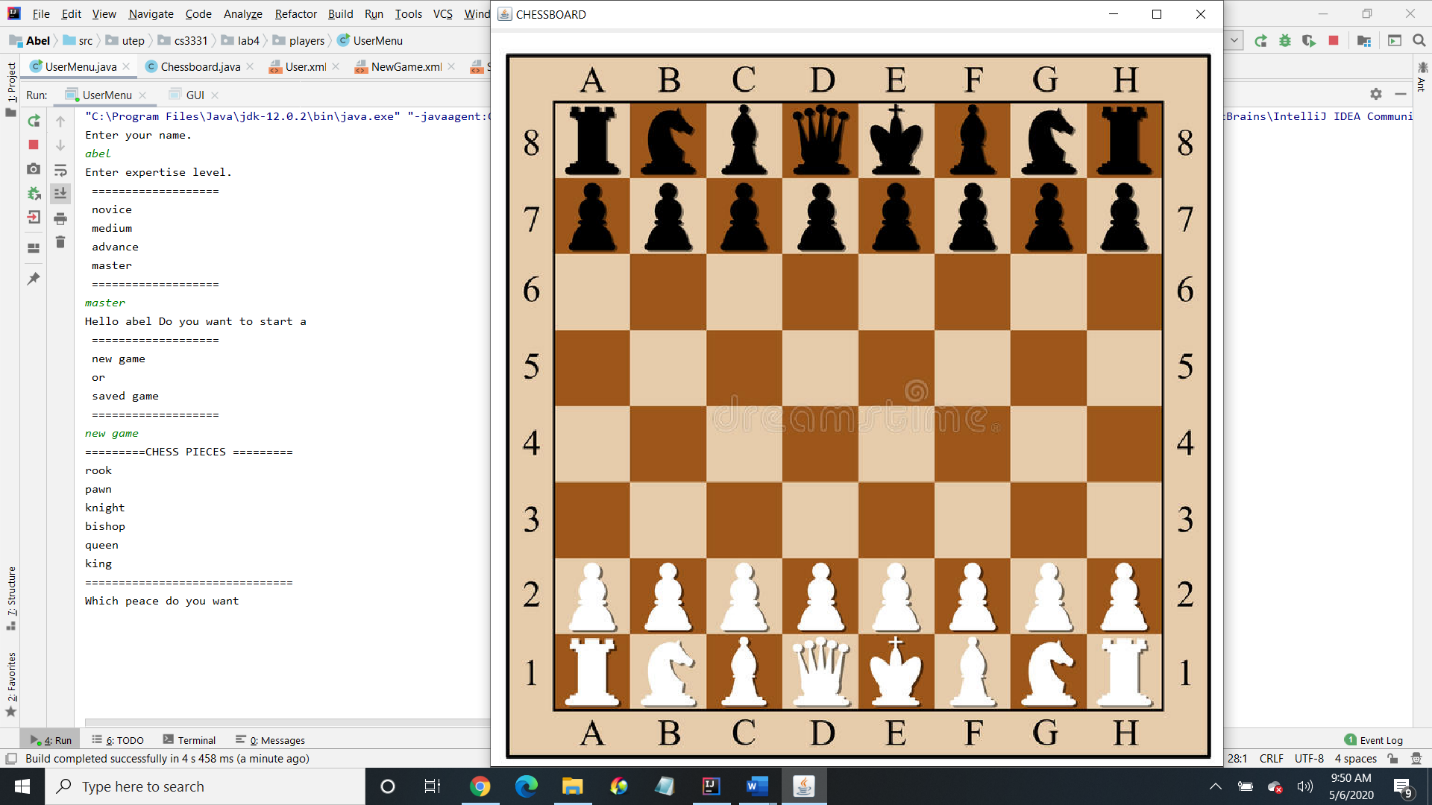
*I selected this method because < I used different movements for pawn and king>*

*Test case 1 <PawnTest >   
< My strategy is to move the black pawn from (a)(5) to (a)(3) which I used in if statement >  
<* *I used this strategy because it is easier to determine the movement for pawn since it can only move upwards, which I check if the initial(y) is equal to 5, then I simply check if the final column is equal to the initial(x), then from there I check if final row is equal to 4 or final row is equal to 3 (since it hasn’t move in this case) >  
< >*

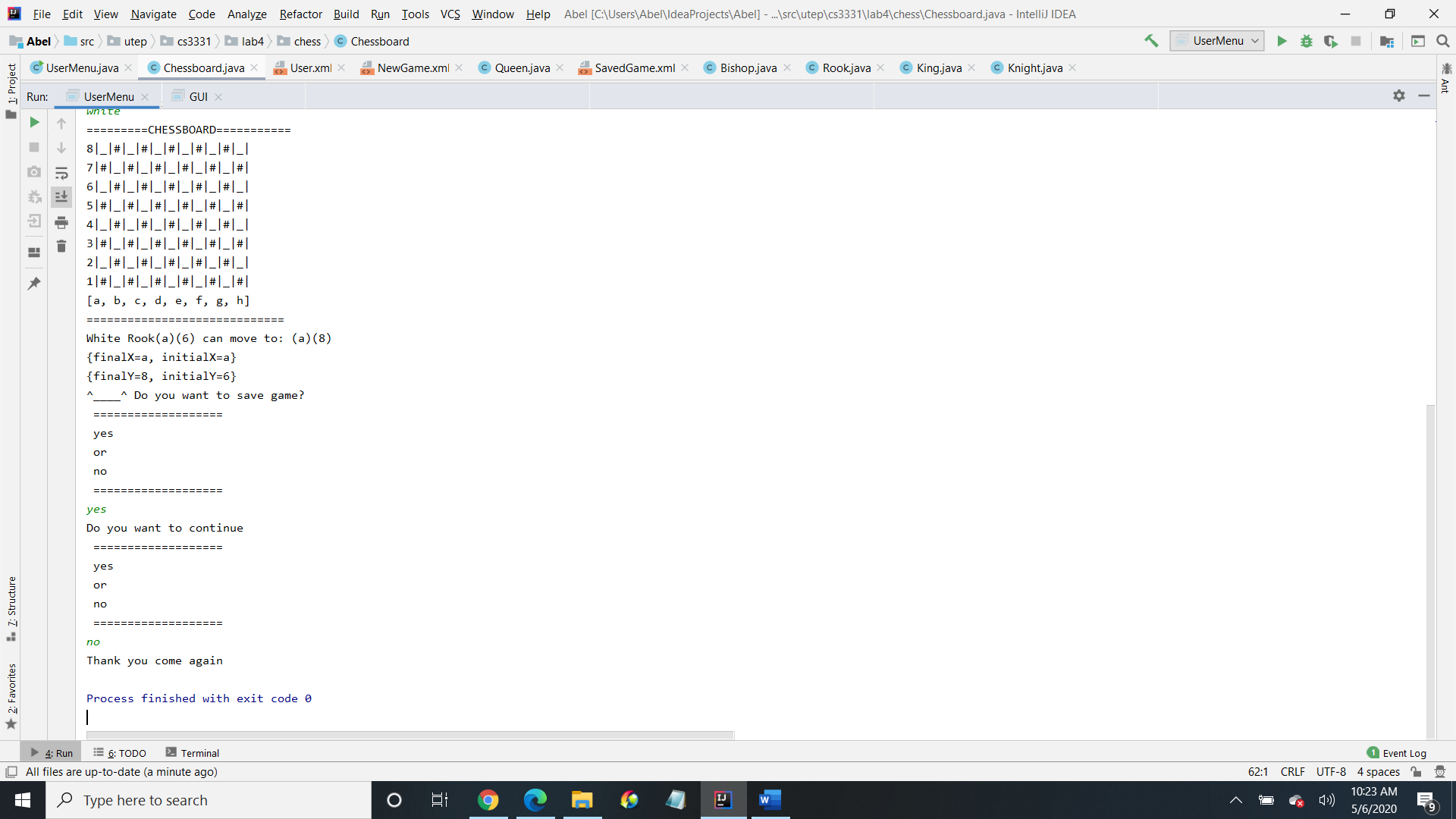
*Test case 2 <PawnTest >   
< My strategy is to move the black pawn from (b)(2) to (b)(1) which I used in if statement >  
< I used this strategy because it is easier to determine the movement for pawn since it can only move upwards, which I check if the initial(y) is equal to 2, then I simply check if the final column is equal to the initial(x), then from there I check if final row is equal to 1 (since it hasn’t move in this case) >  
< >*

*Test case 3 <KingTest>   
< My strategy is to move the white king from (g)(1) to (h)(2) which I used a for loop to move up the right >  
< I used this strategy because the for loop iterates the movement till its correct (incrementing), by adding initial (x) +1 and by checking if its equal to the final column, including initial (y) + 1 if its equal to final row, (which is the kings movement) depending on the initial and final positions of x and y >  
< >*

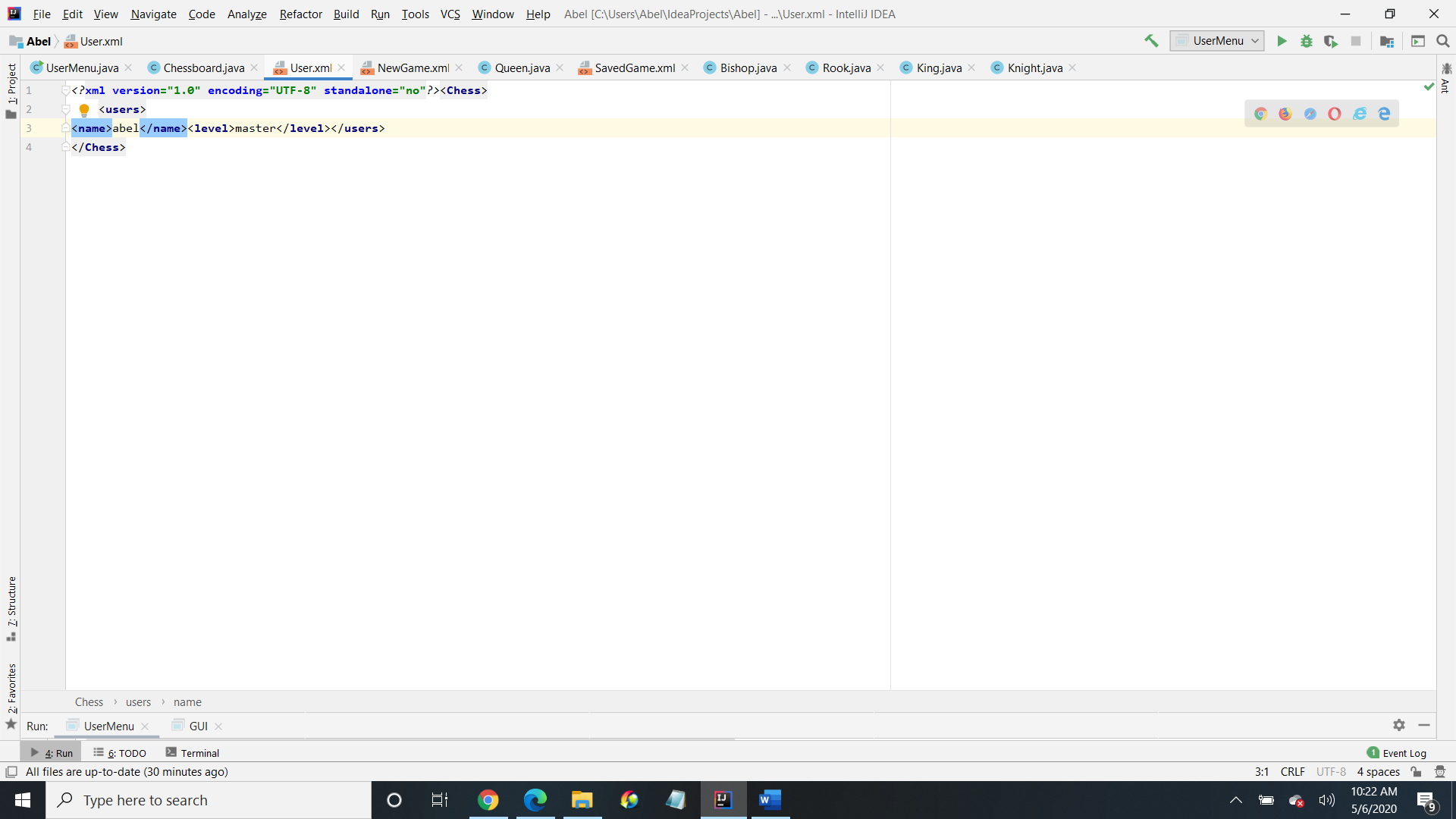
Demonstrate your code functionality –   
 *- I begin by asking the user for their name (I save their name in the XML file), expertise level (I check if it’s a valid level) and save their expertise level in the XML file. then I greet them by asking them if they would like to start a new game (if they pick new game, it is saved in the XML file) or saved game (in addition to the assignment requirements, I added GUI). Once they pick either game, the user then gets to see the GUI that I added of which it is a chessboard. I added the pieces into an array list collection and I used the for-each loop (to go through the list and print out the pieces). After that I ask the user what piece do, they want (I also check if it’s a valid piece), then I ask what color of the piece (I also check if it’s a valid color). I then created some linked lists with the types of character and integer for x and y. Which I then added the characters and integers to x and y into the linked lists. Then I created an iterator for iterating y from which I used a while loop, while y has a next one and then I print each y out vertically, for like a chessboard (ASCII). Then I created an iterator for iterating x from which I iterated each x while it has a next one and printed it out horizontally. Then I have a switch type depending on the user decisions on the type and color of the piece. Each piece is using a strategy pattern from where I am encapsulating each one (e.g., rook at initial (a)(6) to final (a)(8)) and can be interchangeable. Then I created a hash map for each piece, with the types of <String, Character> for X and <String, Integer> for Y. Then I put the strings and characters into X for the hash map, and the strings and integers into Y for the hash map. Then I print each one out. I then aggregate a stream operation with a for-each for determining if it is successful on each piece. Then I ask user if they want to save the game (if yes, the game is saved in the XML file using the piece and color). Then I ask the user if they want to continue (while the decision is yes, the game will continue). If the user doesn’t want to continue, I then thank the user and tell the user to come again (then the game closes). At the end I tested the (queen, knight, pawn, and king). I executed each one by initializing the (color, x, y), then I created a method based on their movements (using strategy pattern) for each piece, and then I printed the test’s out.*



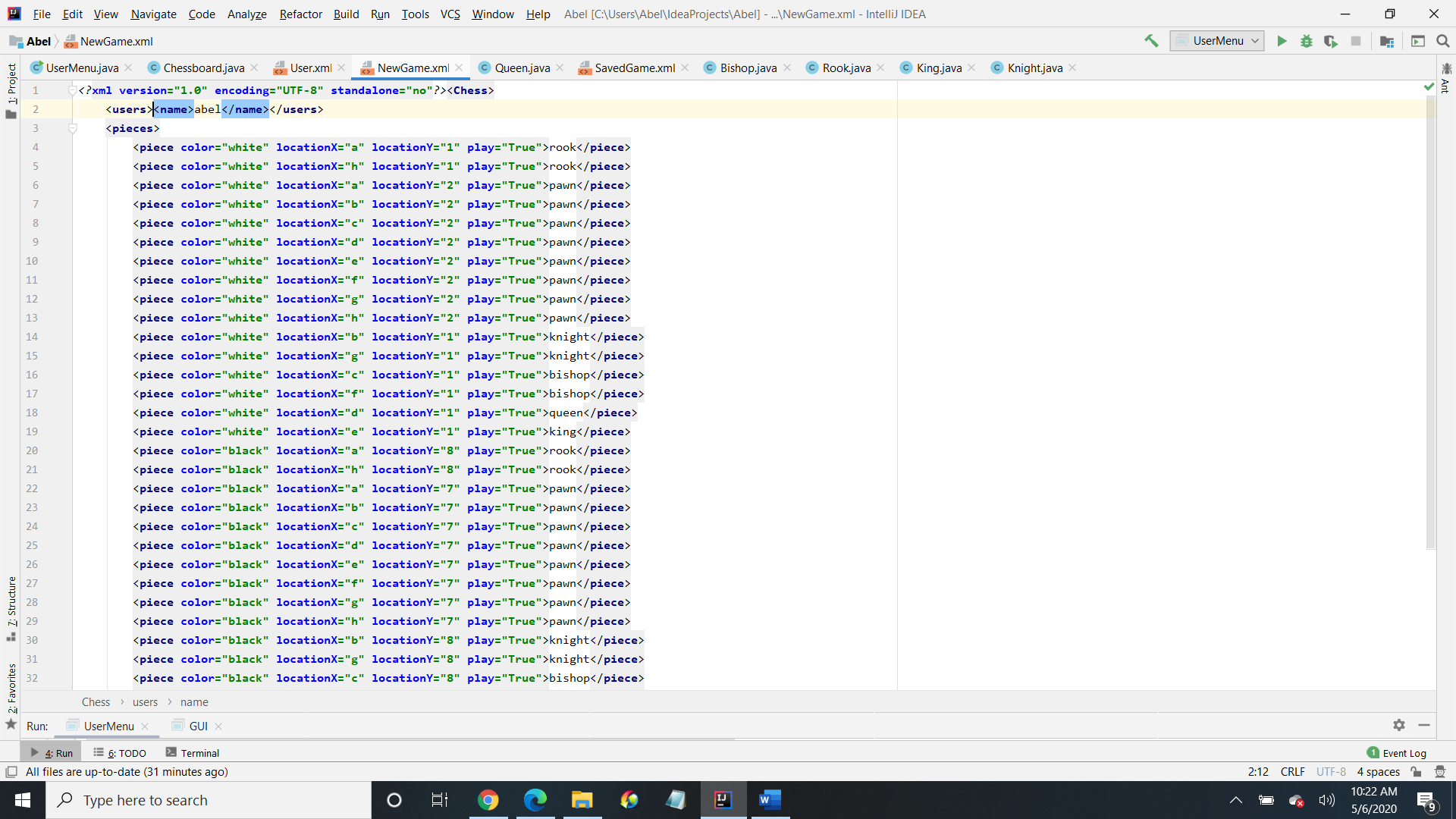
\*Here is where Chess GUI appears\*



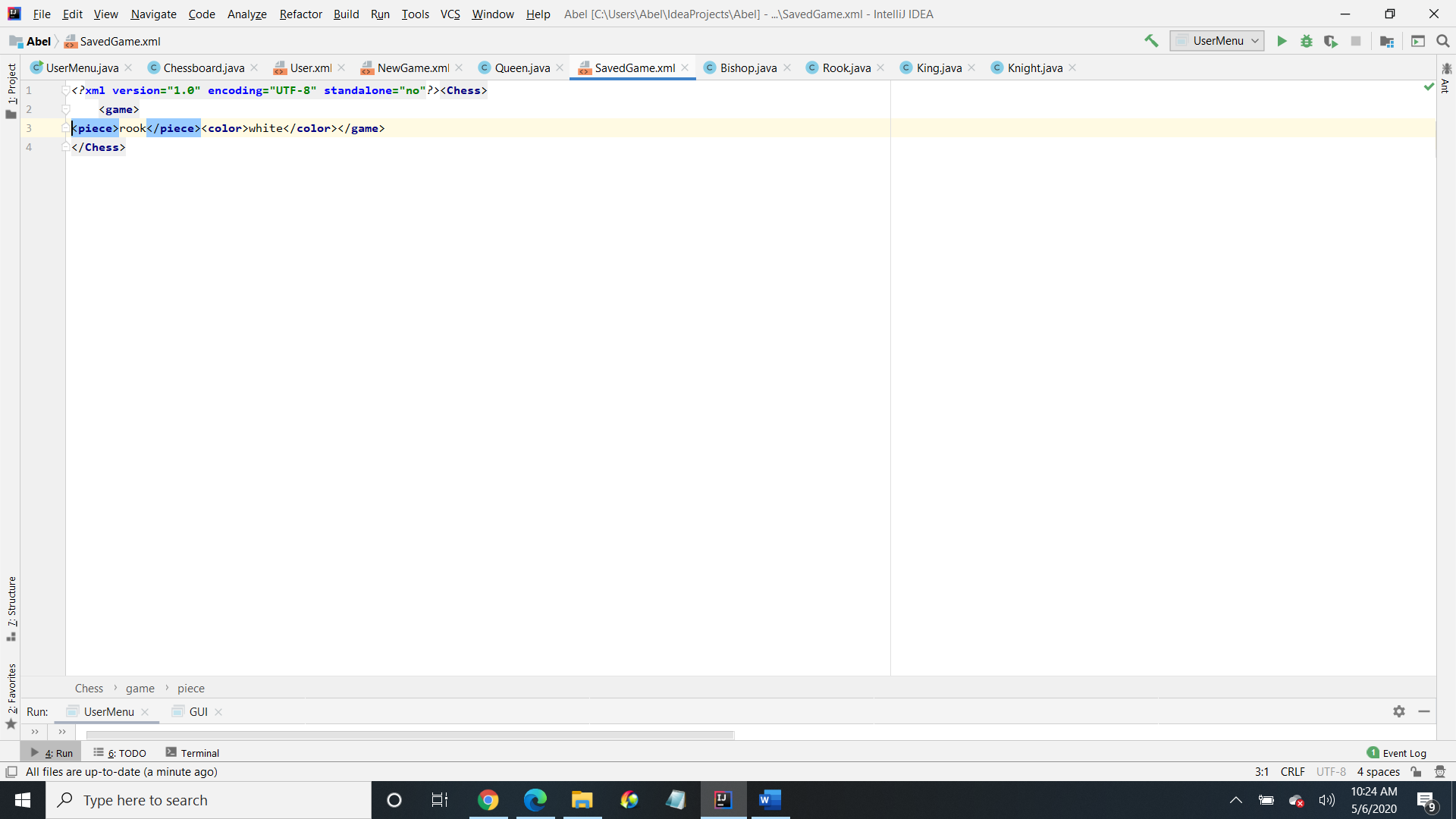
\*Here is the result from where the piece is picked from the user and if the user wants to save the game and continue\*



\*Here is where the user’s name and expertise level are saved (XML)\*



\*Here is where the user saves the name in the new game (XML)\*



\*Here is where the user saves the name and color in the saved game (XML)\*