# Laboratory 2 – Test-Driven Development

# Software Quality Assurance and Testing

Software College, Northeastern University

# Workbook

Student Number:

20175181

Student Name:

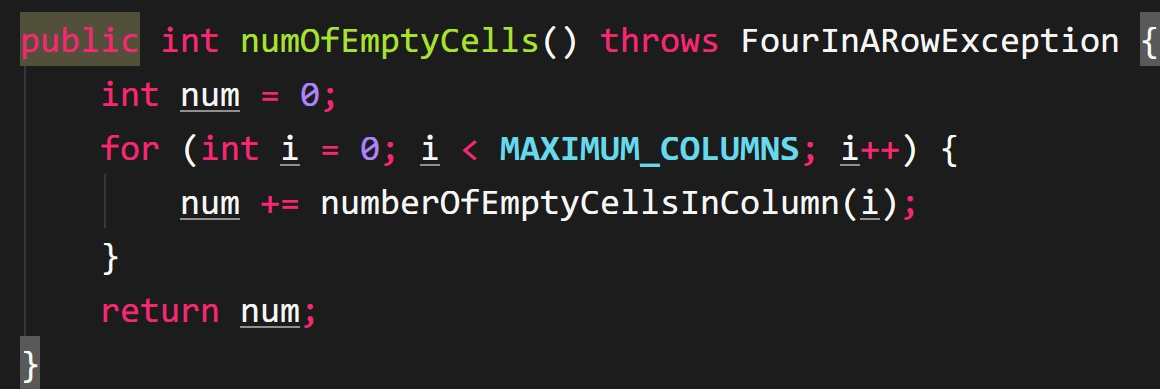
Zenkai Young

When you have finished the technical work, answer the questions in this workbook. The questions are asking you to talk about the work that you did.

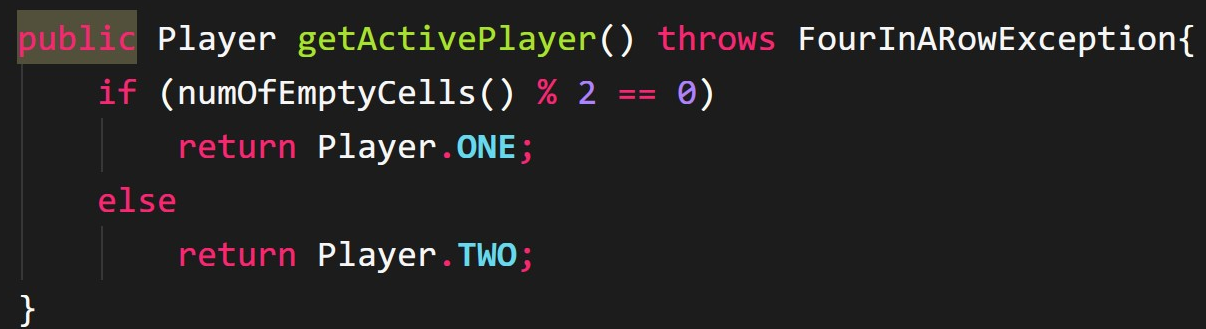
# Question 1

Think about the technical work to write the tests and code for the Game class. Explain the order you worked on the functions in Stage 1. Why did you choose that order?

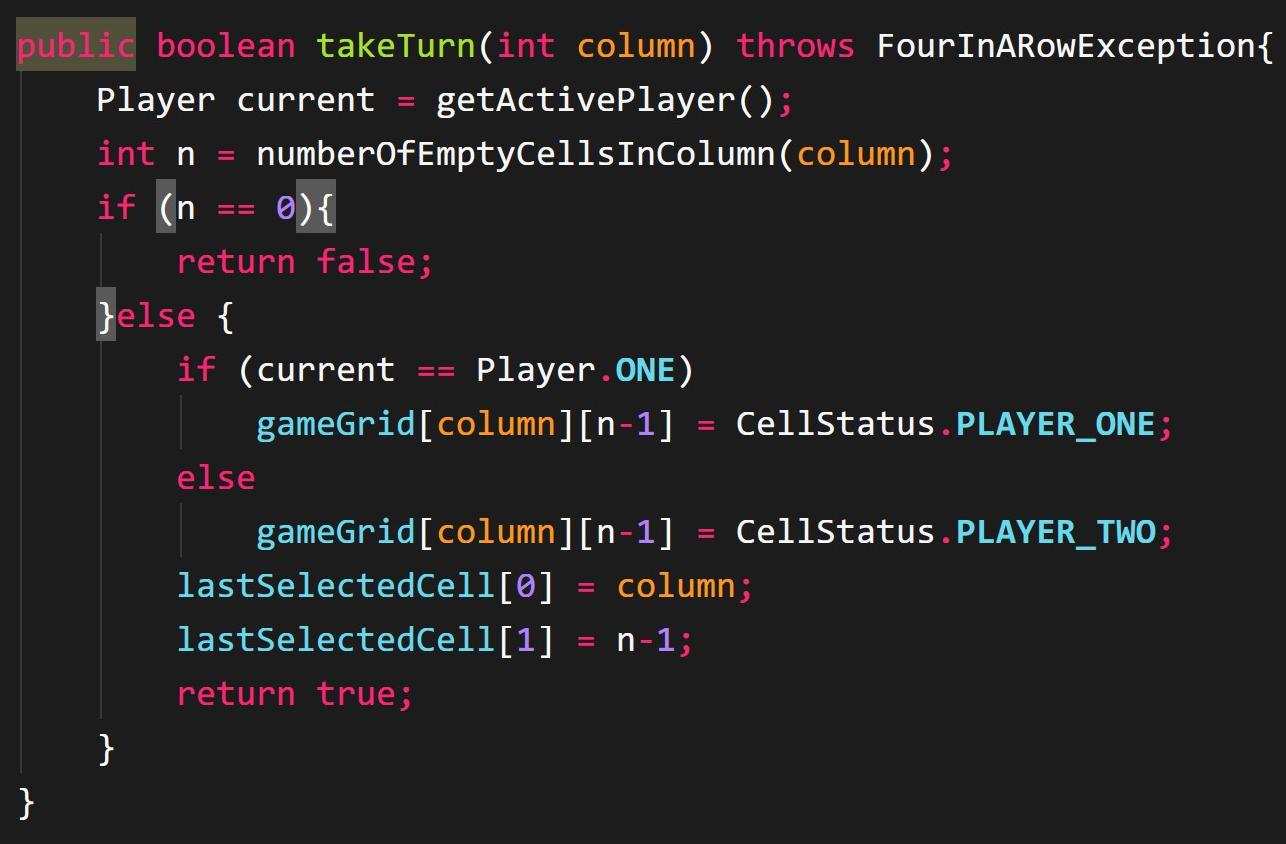
1. First of all, I work on getActivePlayer(). There is a easier way to work on this function which is we can count the number of empty cells in the grid. If the number is even, the active player is player One. On the contrary, if the number is odd, the active player is player Two.(It’s pretty cool right! 😀) So we need a function to count the number of empty cells in the grid. And I notice that we’ve got a function called numberOfEmptyCellsInColumn() which help us to count the number of empty cells in one column. So we can iterate through all the columns to count all the empty cells in the grid like below.



With this function, we can work out getActivePlayer() like below.



1. Secondly, I work on takeTurn(int column). Since we can get active player now, we can work on takeTurn function . First we use numberOfEmptyCellsInColumn() to check if this column has empty cells. If not, we return false. If there are still empty cells, then we update the cell status to current player. And then we record the column and row we have taken for further use.



1. Thirdly, I choose to work on hasWon(). The reason why I didn’t choose to work on resetGame() is that resetGame() need to reset the undo() and redo(). But we don’t have undo() and redo() yet.

There’re three ways to win a game, horizontally win, vertically win and diagonally win. So we split hasWon() into three parts which is horizontallyWin(), verticallyWin() and diagonallyWin().

HorizontallyWin(int col, int row) check if there is a horizontal win and take two parameters which is int col for column to check and int row for row to check. To check a horizontal win, we need to count the left and right continuous cells with the same status as the cell right now. If the sum of two numbers is greater than or equal to 3, we would say there is a horizontal win. And if the sum is less than 3, there is not a horizontal win.

public boolean horizontallyWin(int col, int row){  
 int counterOfLeft = 0;  
 int counterOfRight = 0;  
 CellStatus player = gameGrid[col][row];  
 int numOfLeft = Math.min(3,col); //number of the left cells that should be check  
 int numOfRight = Math.min(3,**MAXIMUM\_COLUMNS**-1-col); //number of the right cells that should be check  
  
 if (player == CellStatus.**EMPTY**)  
 return false;  
 // count the number of cells with the same status in the left  
 boolean leftFlag = true;  
 for (int i = col-1; i >= col-numOfLeft; i--) {  
 if (player == gameGrid[i][row] && leftFlag){  
 counterOfLeft++;  
 }else {  
 leftFlag = false;  
 }  
 }  
 // count the number of cells with the same status in the right  
 boolean rightFlag = true;  
 for (int i = col+1; i <= col+numOfRight; i++) {  
 if (player == gameGrid[i][row] && rightFlag){  
 counterOfRight ++;  
 }  
 else {  
 rightFlag = false;  
 }  
 }  
 return counterOfLeft+counterOfRight>=3;  
}

And we can work out verticallyWin() in a similar way.

verticallyWin(int col, int row) check if there is a vertical win and take two parameters which is int col for column to check and int row for row to check. To check a vertical win, we need to count the up and down continuous cells with the same status as the cell right now. If the sum of two numbers is greater than or equal to 3, we would say there is a vertical win. And if the sum is less than 3, there is not a vertical win.

public boolean verticallyWin(int col, int row){  
 int counterOfUp = 0;  
 int counterOfDown = 0;  
 CellStatus player = gameGrid[col][row];  
 int numOfDown = Math.min(3,row); //number of the down cells that should be check  
 int numOfUp = Math.min(3,**MAXIMUM\_ROWS**-1-row); //number of the up cells that should be check  
  
 if (player == CellStatus.**EMPTY**)  
 return false;  
  
 // count the number of cells with the same status downward  
 boolean downFlag = true;  
 for (int i = row-1; i >= row-numOfDown; i--) {  
 if (player == gameGrid[col][i] && downFlag){  
 counterOfDown ++;  
 }else {  
 downFlag = false;  
 }  
 }  
  
 // count the number of cells with the same status upward  
 boolean upFlag = true;  
 for (int i = row + 1; i <= row+numOfUp; i++) {  
 if (player == gameGrid[col][i] && upFlag){  
 counterOfUp ++;  
 }else {  
 upFlag = false;  
 }  
 }  
 return counterOfDown + counterOfUp >= 3;  
}

And diagonallyWin is a little bit different. We need to check four directions which is up left, up right, down left, down right.

diagonallyWin(int col, int row) check if there is a diagonal win and take two parameters which is int col for column to check and int row for row to check. To check a diagonal win, we need to count the sum of up right and down left and the sum of up left and down right continuous cells with the same status as the cell right now. If either one of these two sum is greater than or equal to 3, we would say there is a vertical win. And if the sum is less than 3, there is not a vertical win.

public boolean diagonallyWin(int col, int row){  
 int counterOfUpleft = 0;  
 int counterOfUpright = 0;  
 int counterOfDownleft = 0;  
 int counterOfDownright = 0;  
 CellStatus player = gameGrid[col][row];  
 int numOfUpleft = Math.min(Math.min(col,3),Math.min(**MAXIMUM\_ROWS**-1-row,3)); //number of the up left cells that should be check  
 int numOfUpright = Math.min(Math.min(**MAXIMUM\_ROWS**-1-row,3),Math.min(**MAXIMUM\_COLUMNS**-1-col,3)); //number of the up right cells that should be check  
 int numOfDownleft = Math.min(Math.min(row,3),Math.min(col,3)); //number of the down left cells that should be check  
 int numOfDownright = Math.min(Math.min(row,3),Math.min(**MAXIMUM\_COLUMNS**-1-col,3)); //number of the down right cells that should be check  
 if (player == CellStatus.**EMPTY**)  
 return false;  
  
 // count the number of cells with the same status upleft  
 boolean upLeftFlag = true;  
 for (int i=row+1,j=col-1;i<=row+numOfUpleft;i++,j--){  
 if (player==gameGrid[j][i]&&upLeftFlag){  
 counterOfUpleft++;  
 }else {  
 upLeftFlag=false;  
 }  
 }  
  
 // count the number of cells with the same status up right  
 boolean upRightFlag = true;  
 for (int i=row+1,j=col+1;i<=row+numOfUpright;i++,j++){  
 if (player==gameGrid[j][i]&&upRightFlag){  
 counterOfUpright++;  
 }else {  
 upRightFlag=false;  
 }  
 }  
  
 // count the number of cells with the same status down left  
 boolean downLeftFlag=true;  
 for (int i=row-1,j=col-1;i>=row-numOfDownleft;i--,j--){  
 if (player==gameGrid[j][i]&&downLeftFlag){  
 counterOfDownleft++;  
 }else {  
 downLeftFlag=false;  
 }  
 }  
  
 // count the number of cells with the same status down right  
 boolean downRightFlag=true;  
 for (int i=row-1,j=col+1;i>=row-numOfDownright;i--,j++){  
 if (player==gameGrid[j][i]&&downRightFlag){  
 counterOfDownright++;  
 }else {  
 downRightFlag=false;  
 }  
 }  
  
 return counterOfUpleft+counterOfDownright>=3 || counterOfUpright+counterOfDownleft>=3;  
}

Above is my first design. After refactoring, I found a way to improve efficiency of my code. Instead of using a flag to control the branch, I chose to break directionly like below

for (int i = row-1,j=col-1; j >= col-numOfUpleft; i--,j--) {  
 if (player==gameGrid[j][i]){  
 counterOfUpleft++;  
 }else {  
 break;  
 }  
}

I came up with this idea when I do refactoring lol😂. It’s easier and more efficient.

As for hasWon() we can call three functions above to check a win and the parameters are the column and row we have recently taken. If either one of the three functions return true, we have a win.

public boolean hasWon(){  
 return horizontallyWin(lastSelectedCell[0],lastSelectedCell[1]) || verticallyWin(lastSelectedCell[0],lastSelectedCell[1]) || diagonallyWin(lastSelectedCell[0],lastSelectedCell[1]);  
}

1. Fourthly, I work on undo(). Since we have recorded the lastSelectedCell, we can just change the status of that cell to EMPTY. And by doing that, the currently active player is swapped to the other player automatically(because our getActivePlayer() got some magic lol😂). And we use a flag called redoFlag to decide if we can redo. After a undo, we set this flag to true which means we can redo after an undo.

public boolean undo(){  
 gameGrid[lastSelectedCell[0]][lastSelectedCell[1]] = CellStatus.**EMPTY**;  
 redoFlag = true; //after a undo, we can redo.  
 return true;  
}

1. And then we can work on redo(). At first we check the redoFlag. If redoFlag is false, we can’t redo because there is no matching undo. If redoFlag is true, we can redo the operation according to the currently active player.

public boolean redo() throws FourInARowException {  
 if (redoFlag){  
 Player p = getActivePlayer();  
 if (p == Player.**ONE**)  
 gameGrid[lastSelectedCell[0]][lastSelectedCell[1]] = CellStatus.**PLAYER\_ONE**;  
  
 else if (p == Player.**TWO**)  
 gameGrid[lastSelectedCell[0]][lastSelectedCell[1]] = CellStatus.**PLAYER\_TWO**;  
 redoFlag = false;  
 return true;  
 }else {  
 return false;  
 }  
}

1. At last, we can work on the resetGame(). First we call reset() to reset the board so that every position is empty. By doing that, active player is reset to Player One. And then we refresh lastSelectedCell to reset undo and redo since undo and redo are based on the lastSelectedCell.

public void resetGame(){  
 reset();  
 lastSelectedCell = new int[2];  
}

# Question 2

Write about your experience of using a Test-First approach on this project. Was it useful or was it a problem? Explain your answer. Write approximately 200 words.

After I got the lab 2 Task Description, I read through it. All I felt is fear and had no idea what to do first.😱 Then I tried Test-First approach. It is really useful. First, it can relieve us from complicated requirements and make us focus on one specific task. And we write tests first. Honestly, I think writing test is easier than writing implementation. After we got the test, we can write an easy version to accomplish the requirements. If test failed, we debug and modify the implementation. If test passed, we can do some refactoring to lessen the redundancy and improve the efficiency of our code. And TDD give us the protection for refactoring which make us improve our code confidently. And writing tests first make us clearer of the requirements and details rather than find some wrong understanding of requirements in the mid-term. And TDD can give us rapid feedback. If we didn’t get some unit tests at first, we still needed to write some tests in the mid-term which will take a lot of time to prepare data. That means wo got rapid feedback if we use TDD.

# Question 3

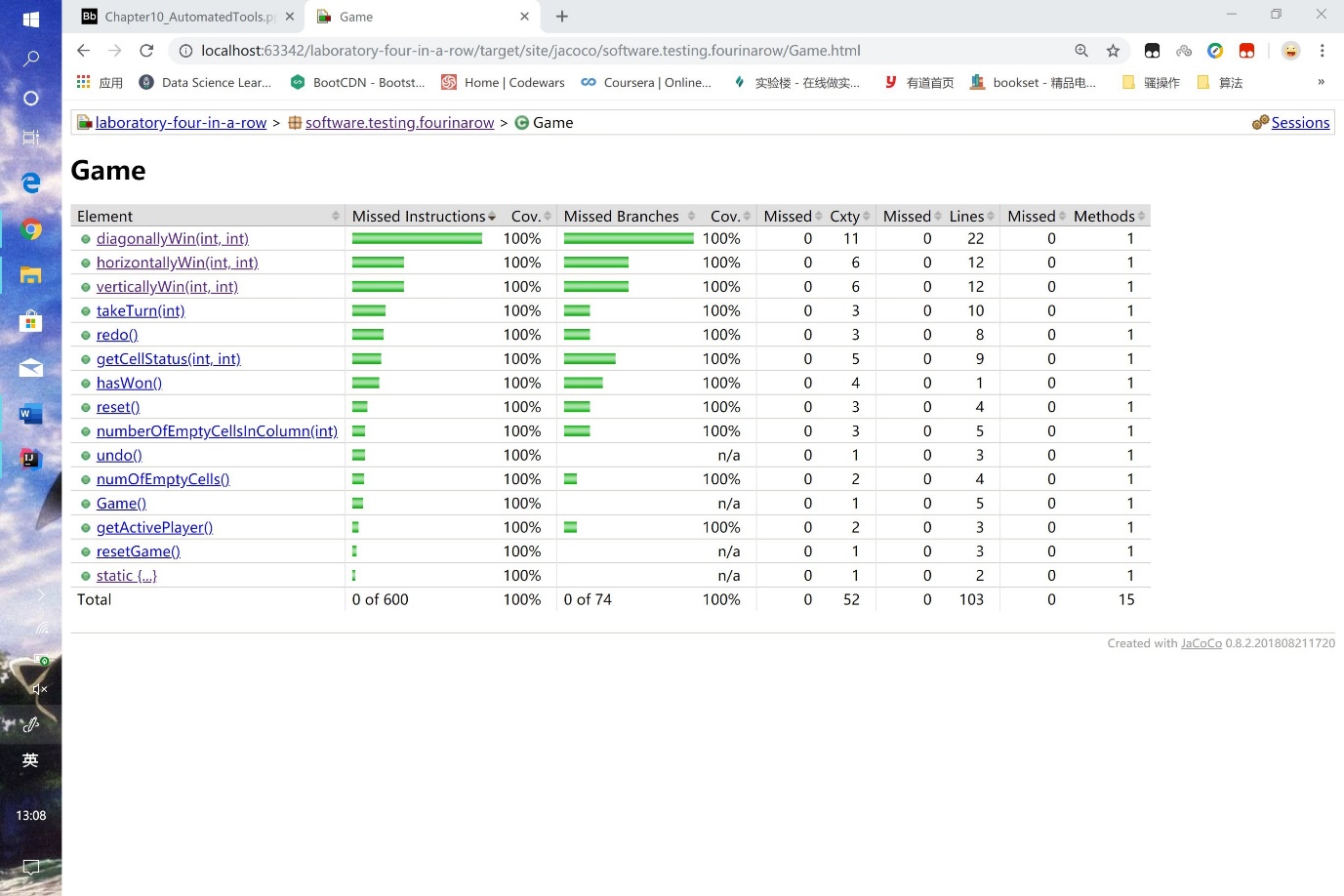
Did you use Refactoring in this work? If yes, explain how you used it.

Yes, I did use refactoring in my work. For example, at first, I got a lot of redundancy in my hasWon() because I put all three checks which are horizontal win, vertical win and diagonal win into one function. And I use nested for loop to achieve this method which is unnecessary and have little efficiency. Then I need to refactor. First I got three check split from hasWon(), and changed the way it achieves which take two parameter(column and row) to improve efficiency. Finally I just got one line in my hasWon() which is cool😂

And after refactoring, I found some problems in my horizontallyWin(), verticallyWin() and diagonallyWin() which is the use of flag. Actually I can just use a break to stop the for loop and it did improve the efficiency of the code and make my code clearer and more readable.

# Question 4

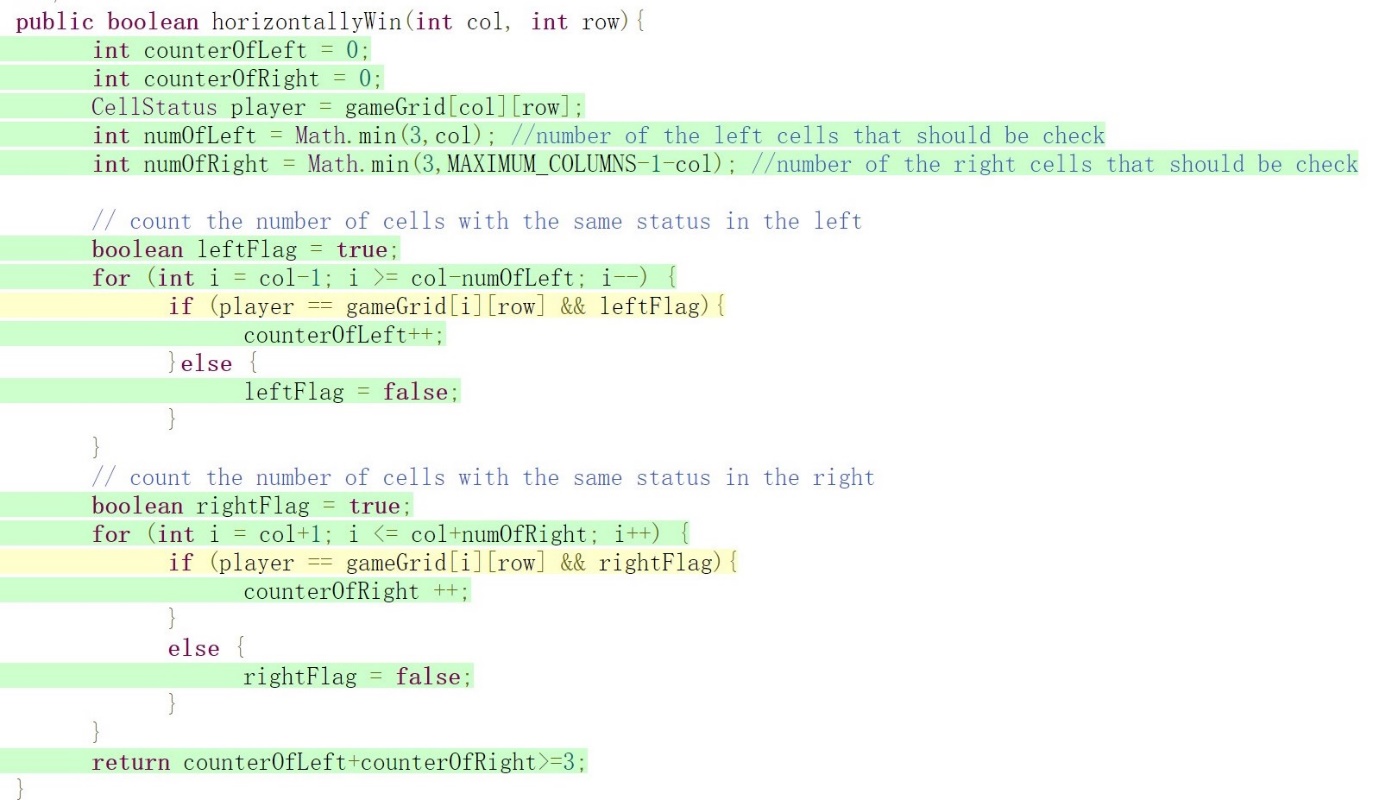
Include a screenshot showing the summary of the code coverage information from JaCoCo. The JaCoCo report will be generated in target/site/jacoco/. Open the index.html file and then navigate to the screen showing code coverage for the Game class. Take a screenshot and include it here.



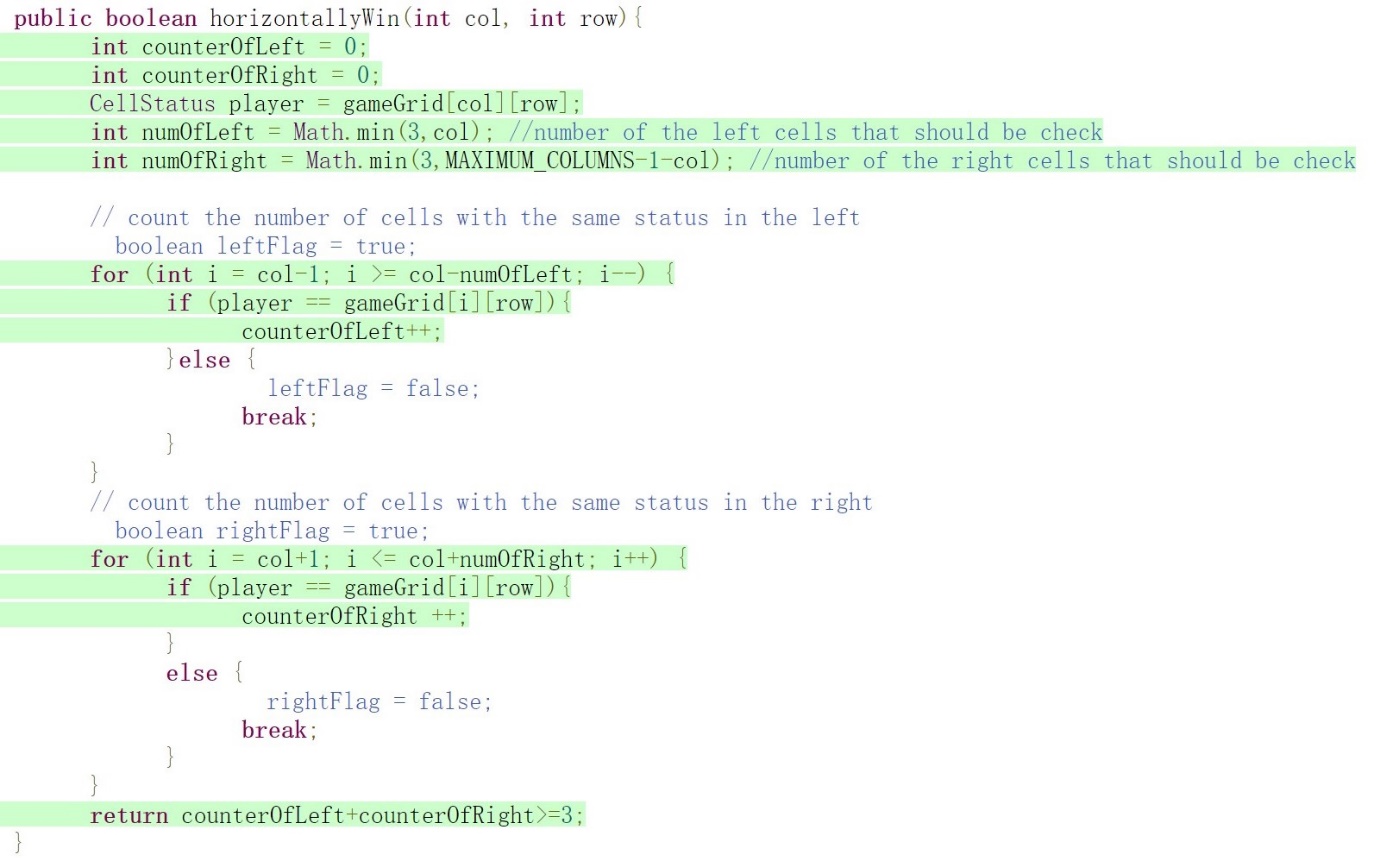
Did you manage to get full code coverage of the instructions and branches? Explain your answer.

Yes. When I work on hasWon(), this method is split into three parts. These three parts have four directions to check and it’s hard to cover all the instructions at first version. So we need to add some test cases to make sure of out full code coverage of the instructions.

And at first, my horizontallyWin() has some problem. There are some branches that just can’t be covered like below.

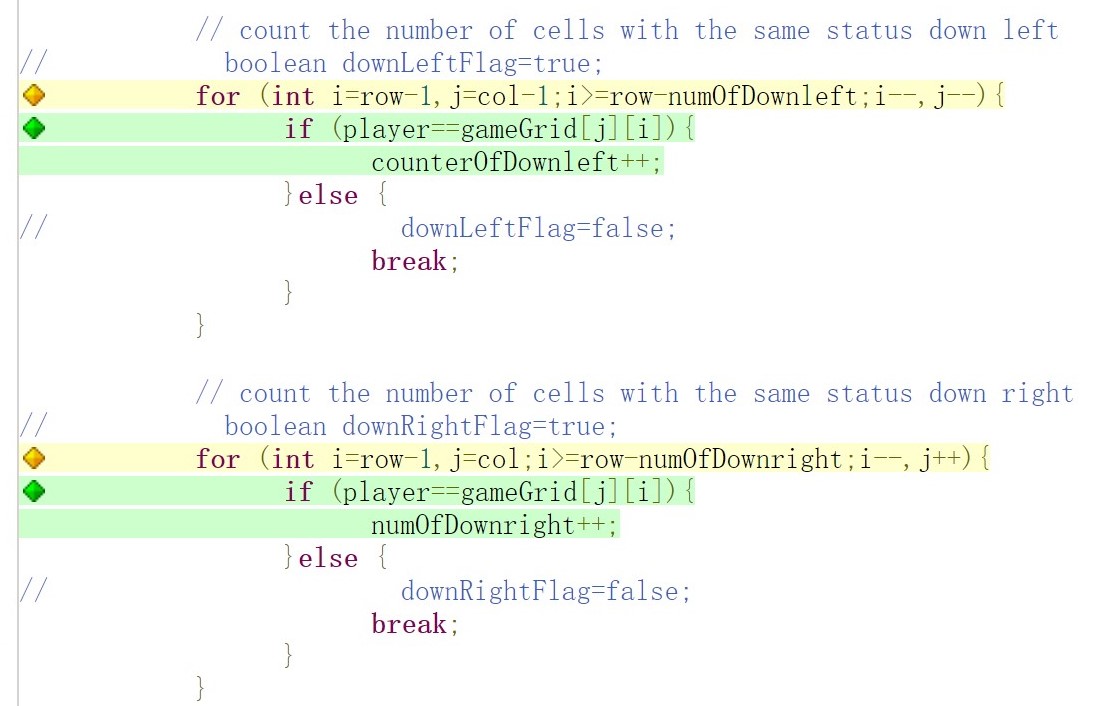


So when I try to cover this branch, I came up with a better idea which also made the code more efficient like below.

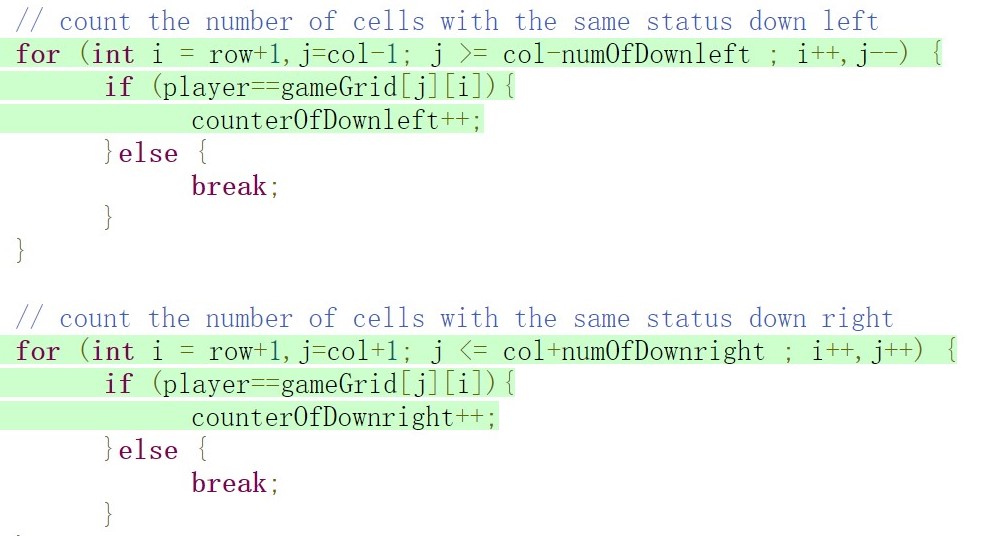


Instead of using a flag to control when to stop, I chose to use a break. It’ s easier and more efficient.

And there are also some issue in my diagonallyWin(). These branches can be covered by adding some test cases.



After some tesing, I successfully cover these branches.



# Question 5

We will discuss Fitnesse in the final lecture. If we installed Fitnesse, how could you use it to provide Acceptance Testing for the Game class? Write up to 200 words about this.

To use Fitnesses to provide acceptance testing. We need to

1. Enter the Fitnesses front page, and we can edit the page name and help text.
2. And then we need to import our Game class by write a line of code that indicated the path of out Game class.
3. Ane then we begin to write our test cases. At first line, we write the name of the parameters and the methods of Game class we need to test. The names is split by “ | ” and we need to add a “ ? ” after the name of method which indicate it’s a method to be tested. At next line and after, we write our inputs of parameters and expected outputs split by “ | ”. After completing, we save the page.
4. At last, we click the test button, Fitnesse will run the tests for us and compare the actual outputs with expected outputs. And if test passes, we get green row. Otherwise, red indicates a test failure 😭!

Fitnesse only need us to write some test cases and expected results. And it will do the test automatically without extra need of test code which is convenient for us.

This is my understanding of Fitnesse and maybe there is wrong opinion. But I will try my best to learn more and more about software testing. And thank you,sir Taylor, for bringing us brilliant lectures of software testing.😆

**Note:** you **do not** need to install or run Fitnesse to do this. You only need to explain what would be necessary to setup for running Acceptance Tests using Fitnesse with the Game class.

# Marking

* 70% of the marks will be for the tests and the code that are produced
* 30% of the marks will be for the answers in this workbook