Harley Phung – CSDS 132 – Case Western Reserve University

CSDS Project 4 Report

- In this project, I create 1 public class called **Tsuro** and 2 nested class that handle the Action Event created in this public class.
- Also, I create 29 helper methods and 1 main method to launch the JavaFX.

REPORT FOR METHODS

1. ***getNumRow():*** this method returns the number of rows in the playing board.

⇨ The method runs successfully. The return matches with the expected value.

2. ***getNumCol():*** this method returns the number of columns in the playing board.

⇨ The method runs successfully. The return matches with the expected value.

3. ***getHandCard():*** this method returns the number of card in each player's hand

⇨ The method runs successfully. The return matches with the expected value.

4. ***getNumPlayers():*** this method returns the number of players participate in Tsuro game.

⇨ The method runs successfully. The return matches with the expected value.

Notes: The followings methods are designed for 2 players only.

5. ***getChosenButton1():*** this method returns the button player 1 choose to implement to the board.

⇨ The method runs successfully. The return matches with the expected value.

6. ***setChosenButton1():*** this method changes the button player 1 choose to implement to the board.

⇨ The method runs successfully. The return matches with the expected value.

7. ***getChosenButton2():*** this method returns the button player 2 choose to implement to the board.

⇨ The method runs successfully. The return matches with the expected value.

8. ***setChosenButton2():*** this method changes the button player 2 choose to implement to the board.

⇨ The method runs successfully. The return matches with the expected value.

9. ***getSelectedBoard1():*** this method returns the button player 1 has just implemented to the board.

⇨ The method runs successfully. The return matches with the expected value.

10. ***setSelectedBoard1():*** this method changes the button player 1 has just implemented to the board.
⇨ The method runs successfully. The return matches with the expected value.

11. ***getSelectedBoard2():*** this method returns the button player 2 has just implemented to the board.
⇨ The method runs successfully. The return matches with the expected value.

12. ***setSelectedBoard2():*** this method changes the button player 2 has just implemented to the board.
⇨ The method runs successfully. The return matches with the expected value.

13. ***getCurrentBlue():*** this method returns the current position of the player 1's stone (which has the color blue) on the tile that was implemented to the board
⇨ The method runs successfully. The return matches with the expected value.

14. ***setCurrentBlue():*** this method changes the current position of the player 1's stone (which has the color blue) on the tile that was implemented to the board.
⇨ The method runs successfully. The return matches with the expected value.

15. ***getCurrentGreen():*** this method returns the current position of the player 2's stone (which has the color green) on the tile that was implemented to the board
⇨ The method runs successfully. The return matches with the expected value.

16. ***setCurrentGreen():*** this method changes the current position of the player 2's stone (which has the color green) on the tile that was implemented to the board.
⇨ The method runs successfully. The return matches with the expected value.

17. ***thisPosition():*** this methods returns the position of the button clicked on the board, which the first index represents the number of row and the second index represents the number of column.
⇨ The method runs successfully. The return matches with the expected value.

18. ***rotateChosenButton():*** this method changes the path of the button that players choose so that it will rotate clockwise 90 degrees.
⇨ The method runs successfully. The return matches with the expected value.

19. ***removeAllStone():*** this method removes all the stone in the button that players do not want stone in
⇨ The method runs successfully. The return matches with the expected value.

20. ***resetChosenButton():*** this method resets the button to default with color white and new pathways with default stone location
⇨ The method runs successfully. The return matches with the expected value.

21. ***stoneCollide():*** this method checks if the 2 stones of the 2 players meet. If true then game over with a tie. Else continue the program.

⇨ The method runs successfully. The return matches with the expected value.

22. ***outOfBoard():*** this method checks if the stone is out of board. If true then game over, the stone still on the board win.

⇨ The method runs successfully. The return matches with the expected value.

23. ***addChosenButtonToBoard():*** this method implements the player's chosen button to board and reset the chosen button at the same time. If implement successful then increment the number of turn.

⇨ The method runs successfully. The return matches with the expected value.

24. ***findNextTile():*** this method checks if the next tile where the stone supposed to go has pathways. Return the next position of the stone in the next tile

⇨ The method runs successfully. The return matches with the expected value.

25. ***travelStone():*** this method will make the stone go on the path until
    + There's no more paths.
    + The stone go out of board
    + The stone hits the other stone.

⇨ The method runs successfully. The return matches with the expected value.

26. ***gameOver():*** this method resets all fields, and print the line to determine who win and exit the system.

⇨ The method runs successfully. The return matches with the expected value.

27. ***player1():*** this method creates a new stage and grid pane for player 1.

⇨ The method runs successfully. The return matches with the expected value.

28. ***player2():*** this method creates a new stage and grid pane for player 2.

⇨ The method runs successfully. The return matches with the expected value.

29. ***start():*** this method start the JavaFX program with the primary stage containing the board.

⇨ The method runs successfully. The return matches with the expected value.

30. ***main():*** this method launches the program and have the command line arguments for the user to control using Interaction Pane.

⇨ In the Interaction Pane
    + When I type "java Tsuro", the program runs with 6x6 grid, hand size is 3, and number of players are 2.

+ When I type “java Tsuro 4 8”, the program runs with 8x4 grid, hand size is 3, and number of players are 2.

+ When I type “java Tsuro 8 4 2”, the program runs with 4x8 grid, hand size is 2, and number of players are 2.

+ When I type “java Tsuro 5 4 4 5”, the program runs with 5x5 grid, hand size is 4, and number of players are 4. (Since the number of players designed for the program is 2, to check the validity of this command line, I have to call getNumPlayers() method).

REPORT FOR NESTED CLASS AND GAME PLAY.

1. ***PlayerAction():*** this nested class will handle the action events happening on player 1 and player 2 stages. Action included:

- In the user’s turn, that user can only choose card from their hand.
- If the user chooses one of the cards, that card must be highlighted with color Yellow.
- If the user wants to rotate, the card’s path must rotate 90 degrees clockwise and still keep color yellow, the stone must be on position 6 of the tile.
- If the user wants to choose other button, the new card must be highlighted with color yellow, the previous card changes to color white.
- If the user turn’s end (meaning the user has implemented their card to the board), the user cannot access to their hand’s card.

⇨ The program runs successfully.

2. ***BoardAction():*** this nested class will handle the action events happening on the board. Action included.

- If it is the first turn of the player 1, they can only implement their card to the first Column of the board.
- If it is the first turn of the player 2, they can only implement their card to the last Column of the board.
- After the button is implemented:
  + The player’s hand card will change to random pathways where the player has just put into the board, the color of this new button changes to color white, the player’s stone will be add to exact position.
  + The pathways of the chosen button will be copy to the board with the stone. The stone will go to the end of the paths using ***travelStone()*** method.
  + If the stone is out of board or two stone hits, the ***gameOver()*** method is called. The system will print the game status and exit the program.

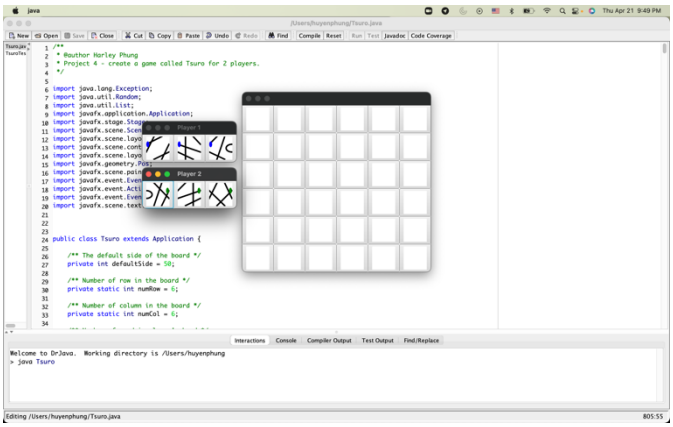
⇨ The program runs successfully.

Example 1:

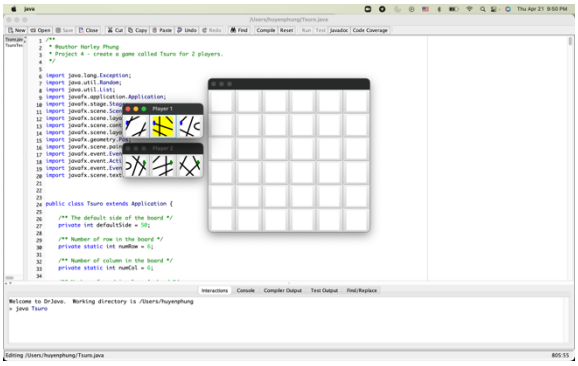
1. Step 1: Call “java Tsuro” in the Interaction Pane.

⇨ The 6x6 board appear with 2 players, each has 3 cards.

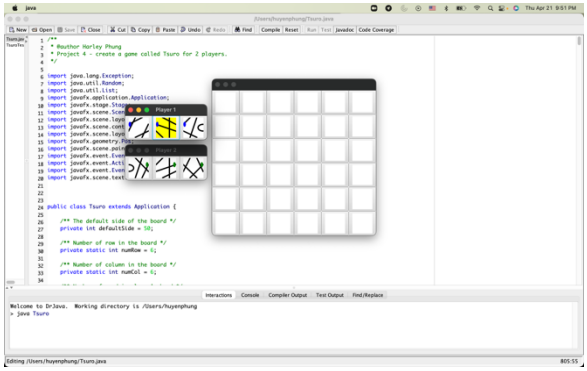
⇨ ***player1(), player2(),*** and ***start()*** method is correct



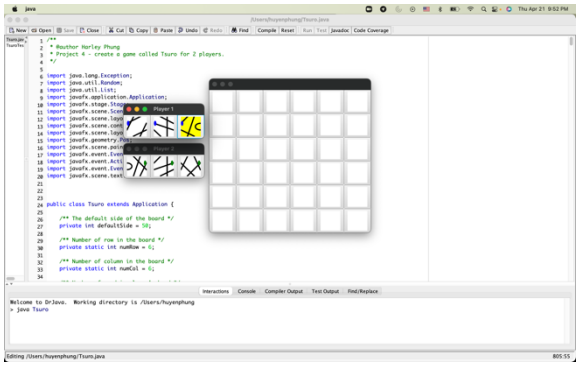
2. Step 2: Choose a button, then rotate and choose another button in player’s stage



Choose the 2nd button, the button is highlighted.



Rotate 180 degrees



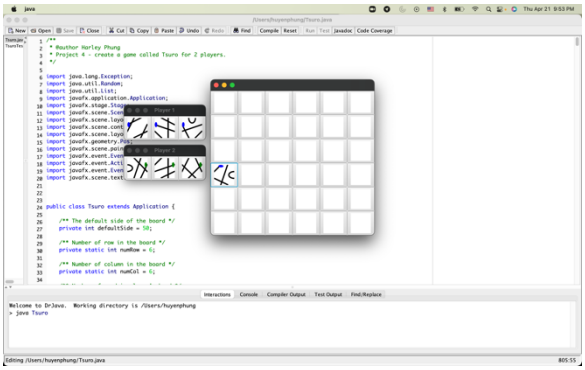
Choose another button, the new button is highlighted, and the old button has color white.

⇨ When it’s player 1, player 2’s hand cannot be clicked.

⇨ The ***rotateChosenButton()*** method, the ***removeAllStones()*** method are correct.

⇨ ***PlayerAction*** handle the action of player 1 is correct.

3. Step 3: Implemented the chosen button to the board.

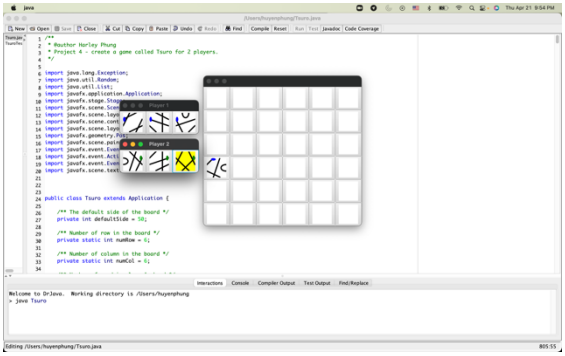


(Notes: The index of the columns from left to right is from 0 to 5, the index of the rows from top to bottom is 0 to 5).

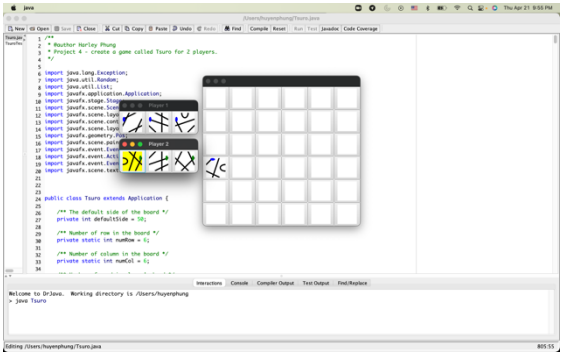
The button is implemented to row 3, column 0. The stone now goes to the position 0, which is the endpoint of the original stone position.

- ⇨ If the player 1 does not choose any button before click to the board, then nothing happens.
- ⇨ If the player 1 choose button different from column 0 of the board, then the chosen button is not implemented, and the player can still choose other buttons to implement. Only when the player 1 choose the right index, the button is implemented, the number of turns is changed.
- ⇨ The 3rd button in player's hand now has different paths
- ⇨ ***resetChosenButton()*** method is correct.
- ⇨ ***BoardAction*** for the first move of the player 1 is correct.

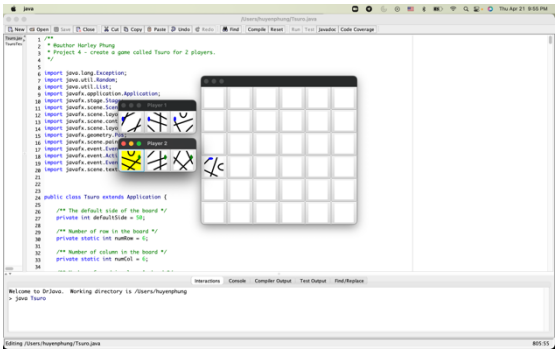
4. Step 4: Choose a button for player 2 to implement.



Choose the 3rd button, it is highlighted

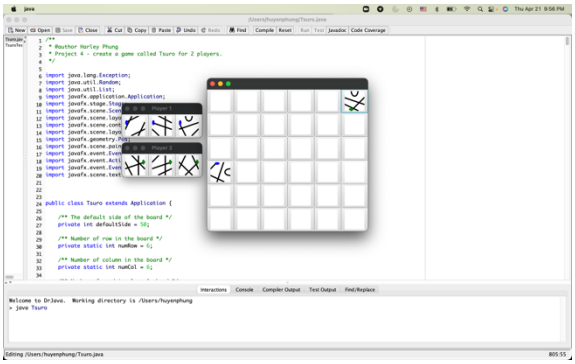


Choose the 1st button, it is highlighted.
The 3rd button is now white



Rotate the 1st button 90 degrees.

5. Step 5: Implement the chosen button to the board



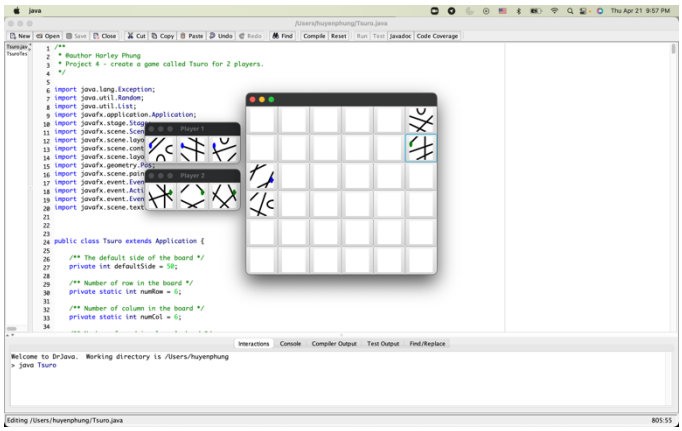
(Notes: The index of the columns from left to right is from 0 to 5, the index of the rows from top to bottom is 0 to 5).

The button is implemented to row 0, column 0.The stone now goes to the position 4, which is the endpoint of the original stone position.

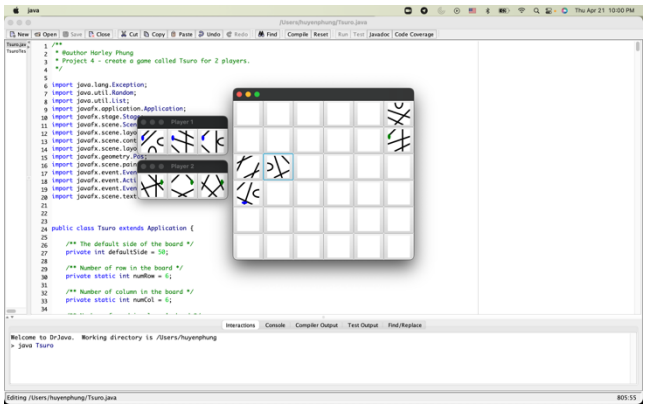
- ⇨ If the player 2 does not choose any button before click to the board, then nothing happens.
- ⇨ If the player 2 choose button different from column 0 of the board, then the chosen button is not implemented, and the player 2 can still choose other buttons to implement. Only when the player 2 choose the right index, the button is implemented, the number of turns is changed.
- ⇨ The 1st button in player's hand now has different paths
- ⇨ ***resetChosenButton()*** method is correct.
- ⇨ ***BoardAction*** for the first move of the player 2 is correct.

6. Step 6: Choose and implement the next stones to the board.
    a. Normal situation.

- ⇨ The buttons for both players are successful implemented, the stones are now move on the right path.
- ⇨ ***thisPosition()*** method, ***resetChosenButton()*** method, ***addChosenButtonToBoard()*** method is correct



b. If there's button that the stone has to go through existed paths.

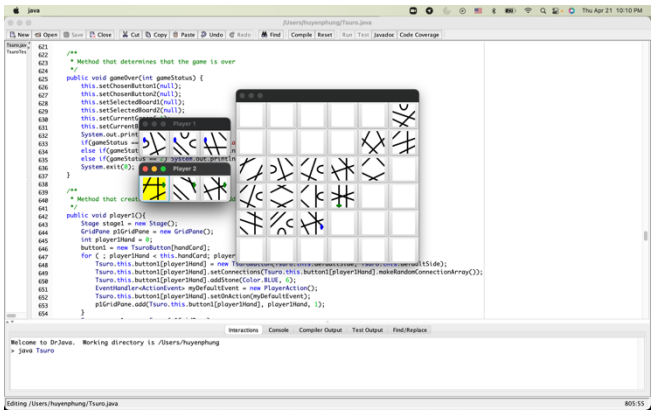


(Notes: if I mentioned 5 - 3 then the it should be understand as
row index - column index)

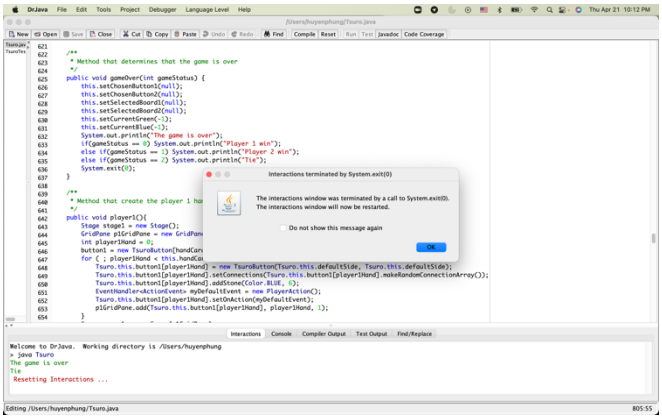
Previously, the stone is in position 2, tile 2 – 0. After placing new path that makes the stone goes back to tile 2 – 0 and tile 3 – 0, it goes to position 4, tile 3 – 0.

- ⇨ The ***thisPosition(), rotateChosenButton(), resetChosenButton(), addChosenButtonToBoard(), findNextTile(), travelStone()*** methods are correct.
- ⇨ The BoardAction and PlayerAction are correct.

7. Step 7: For the sake of the test, my intention is to make the stones collide.



After continuously implement the buttons from players to the board, if I implement the highlighted button to tile 4 – 3, I will have a tie.
The system will print:
"The game is over"
"Tie"
And the Interaction Pane will be reset.



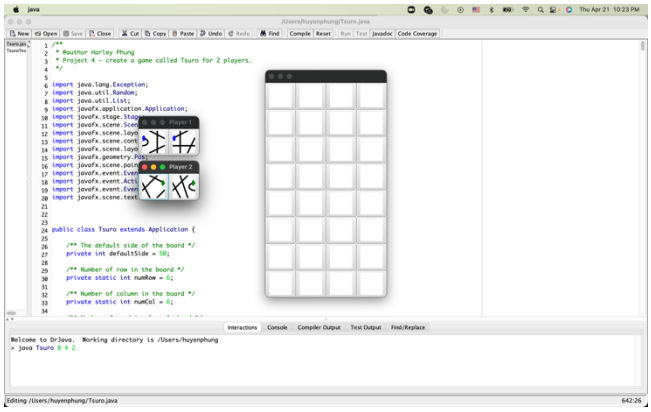
⇨ Aside from previously mentioned methods that is proved to be correct, ***stoneCollide()*** and ***gameOver()*** methods are correct.

Example 1 is finished, all the helper methods except outOfBoard() method that is listed in **REPORT FOR METHODS** section is correct.

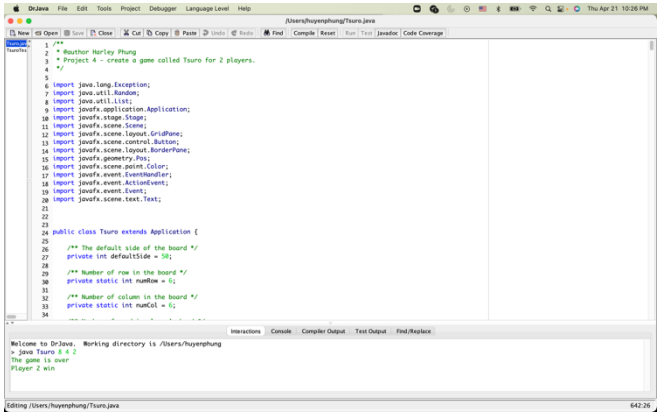
Example 2: In this example, I will not test all the steps like example 1 since all the helper methods except outOfBoard() method are correct. In this example, I mainly test the case that can be used to prove outOfBoard() method is correct. Also, I will call “java Tsuro 8 4 2” in the Interaction Pane to prove the main() method is correct.

1. After calling “java Tsuro 8 4 2”, the board’s grid is 8x4. There are 2 players, each has 2 cards in hand.

⇨ The **main()** method is correct



2. In this situation, if I implement the 1st button in player 1’s hand, the system will print:
   “The game is over”
   “Player 2 win”
   And the Interaction Pan will reset.

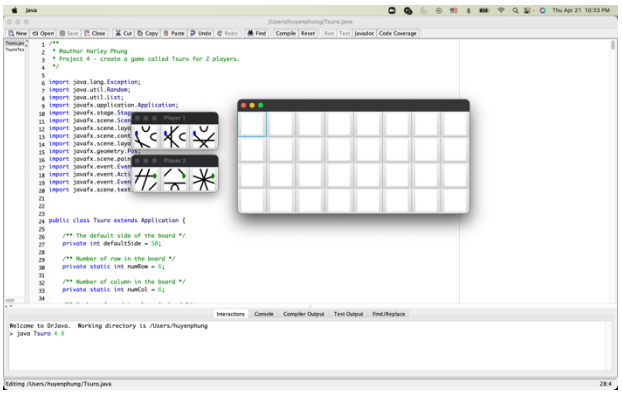


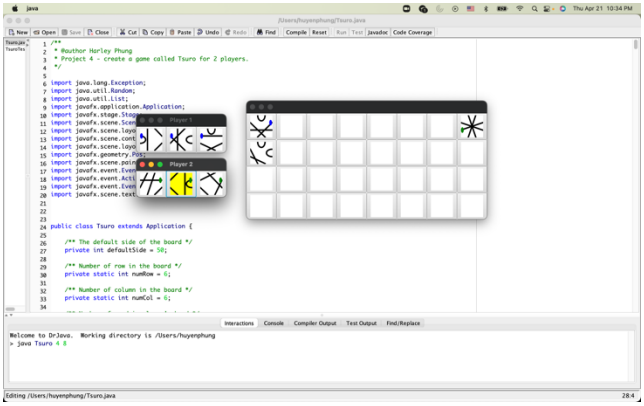
⇨ The ***resetChosenButton(), addChosenButtonToBoard(), outOfBoard()*** and ***gameOver()*** method are correct.

Example 2: In this example, I will not test all the steps like example 1 since all the helper methods except outOfBoard() method are correct. In this example, I mainly test the cases that can be used to prove outOfBoard() method is correct. Also, I will call “java Tsuro 4 8” in the Interaction Pane to prove the main() method is correct.

1. After calling “java Tsuro 4 8”, the board’s grid is 4x8. There are 2 players, each has 3 cards in hand.

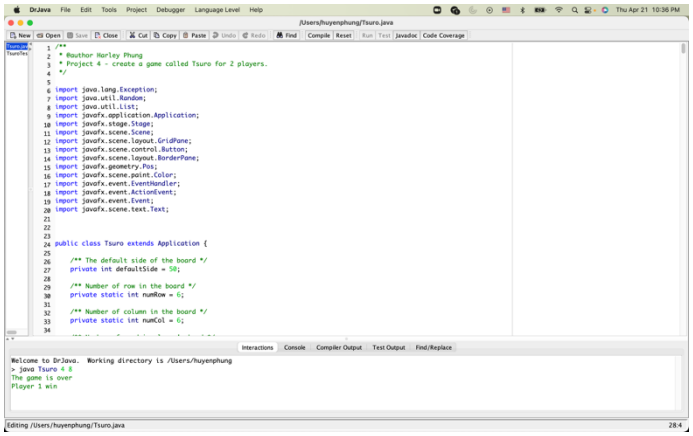
⇨ The **main()** method is correct.





2. In this situation, if I implement the highlighted button in player 2’s hand to tile 0 – 6, the system will print:
“The game is over”
“Player 1 win”
And the Interaction Pan will reset.

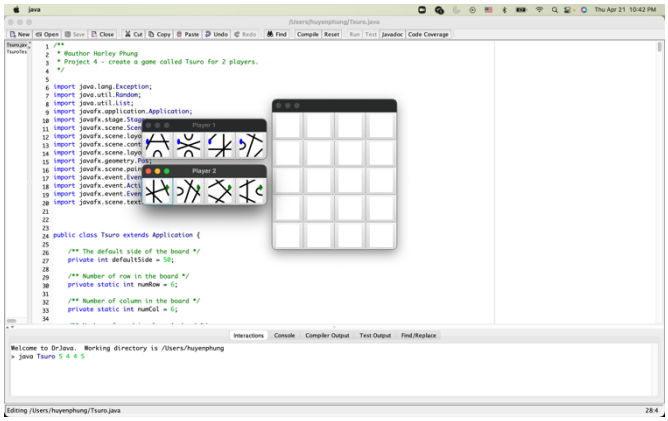
⇨ The ***thisPosition(), resetChosenButton(), addChosenButtonToBoard(), findNextTile(), travelStone()***, ***outOfBoard()*** and ***gameOver()*** methods are correct.



Example 4: In this example, I will play a longer game but still end game by letting the stone out of board. With this example, all the methods except ***stoneCollide()*** is proved to be corrected.

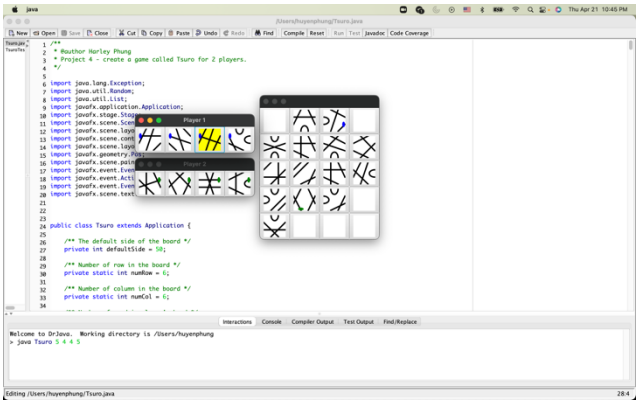
1. After calling “java Tsuro 5 4 4 5” the board’s grid is 5x4. There are supposed to be 4 players, but since the program is designed for 2 players, only 2 players stage showed. Each player’s hand has 5 cards.

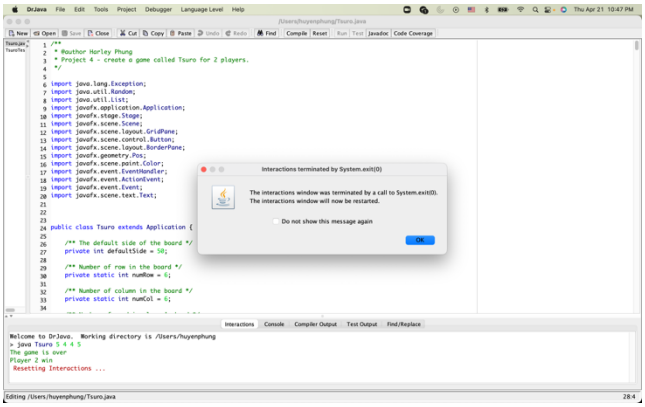
⇨ The **main()** method is correct.



2. All moves for player 1 is now lead player 1's stone to be out of board, which means the game is over for all player's 1 choice. The system will print:

"The game is over"
"Player 2 win"
And the Interaction Pane will reset.





Example 4 is an elaborate case that can prove all methods except for ***stoneCollide()*** method is correct.

SUMMARY

- Because a lot of helper methods cannot be tested using JUnit in DrJava, they are proved to be correct during the playing process and example listed.
- All methods are proved to be correct using JUnit and Cases Study.