Project Deliverable 6—Individual:  
Personal Software Process & Quality

PSP2, Unit Testing—30 points

This assignment must be done by each student individually. No collaboration is allowed.

# Submission Instructions:

Submit a zipped folder named: {YourASURiteUserID}-ProjectDeliverable6.zip

(e.g., skbansa2-ProjectDeliverable6.zip)

This compressed folder should contain the following:

1. Source Code files:
   1. Source code files from Deliverable 4 or 3
   2. Unit test code (placed in the correct package)
2. A generated code coverage report
3. ProjectDeliverable6.docx (or pdf) with Completed Time Log, Estimation worksheet, Design form, Defect Log, Personal Code Review and Project Summary provided at the end of this assignment description.
   1. Make sure to provide responses to the [reflection questions](#_Reflection_Questions_1) listed in ProjectDeliverable6 file (this document).
4. A few screen shots showing
   1. results of your JUnit testing,
   2. code coverage report.
5. A readme file (optional; submit if you have any special instructions for testing).

# Grading Rubric:

Unit Testing—15 points

Test Results, Code coverage report, Postmortem reflection question responses—5 points

PSP process—10 points  
Time log (2), Defect log (2), Estimating Worksheet & Design form (2), Code Review (2), Project Summary (2)

# Program Requirements:

Using JUnit, conduct unit testing for the game logic class(es) and computer player class in the core package. Create a separate package called test for your test class(es).

Use your code from Deliverable 3 or 4. You do not need to test JavaFX class.

Provide test methods for all important methods of the game logic and computer player modules; use equivalence portioning to pick test cases. You should have both success and failure cases. Use a code coverage tool, such as EclEmma, to generate a code coverage report; generate this report for only the core classes you’re placing under test. You must achieve at least 90% code coverage of your game logic and computer player classes. Include the code coverage report in your submission.

# Personal Process:

Follow a good personal process for implementing this game. You will be using PSP2 in this assignment. Estimate and track your effort and defects for the unit testing code that you write. Don’t forget to conduct a personal code review.

## PSP Forms

* Please use the **estimating worksheet** contained herein to estimate how big your program might be.
* Please include in the **design form** any materials you create during your design process. It’s at the end of this document.
* Please use the **code review checklist** contained herein to statically analyze your code for common mistakes.
* Please use the **time log** (provided at the end of this document) to keep track of time spent in each phase of development.
* Please use the **defect log** (provided at the end of this document) to keep track of defects found and fixed in each phase of development.
* When you are done implementing and testing your program, complete the **project summary** form to summarize your effort and defects. Also answer the [reflection questions](#_Reflection_Questions_1).

## Phases

Follow these steps in developing the game:

Plan—understand the program specification and get any clarifications needed.

1. **Estimate** the **time** you are expecting to spend on unit testing.
2. **Estimate** the **defects** you are expecting to inject in each phase.
3. **Estimate** the **size** of the program (only for **new code** that you will be adding).
4. Enter this information in the **estimation columns** of the project summary form. Use your best guess based on your previous programming experience.
5. Use the provided **estimating worksheet** to show how you are breaking up code into smaller modules and estimating.

Design—design the test classes and methods. Test case generation for various test methods needs to be done. Keep track of time spent in this phase and log it. Also keep track of any defects found and log them. Use this phase to design various test methods and test cases.

Code—implement the tests. Keep track of **time** spent in this phase and log. Also keep track of any **defects** found and log them.

CodeReview—use the **code** review **guidelines** provided later in the document to conduct a personal review of your code and fix any issues found. Provide comments in the checklist about your findings. There should be a minimum of 4 comments.

Test—test your program thoroughly and fix any bugs found. The goal here is to test all core classes thoroughly. Keep track of **time** spent in this phase and log. Also keep track of any **defects** found and log them.

Postmortem—complete the actual, and to date **columns** of the **project summary** form and answer the [**reflection questions**](#_Reflection_Questions_1).

Estimating Worksheet

# PSP2 Informal Size Estimating Procedure

1. Study the requirements.
2. Sketch out a crude design.
3. Decompose the design into “estimatable” chunks.
4. Make a size estimate for each chunk, using a combination of:
   1. visualization.
   2. recollection of similar chunks that you’ve previously written
   3. intuition.
5. Add the sizes of the individual chunks to get a total.

## Conceptual Design (sketch your high-level design here)

Test Logic

* Test is valid method
* test the getboard method
* Test MoveX and MoveY Method
* Test Get piece method
* Test Get x value
* Test toString
* test capture method
* test size mthod

Test Computer player

* test move
* test move left
* test move right

Test Wining condition

* test if player 1 wins
* test if player 2 wins
* test if player 1 and player 2 win (tie)

## Module Estimates

|  |  |
| --- | --- |
| Module Description | Estimated Size |
| Test Logic | 300 |
| Test Computer Player | 100 |
| Test Win | 100 |
|  |  |
|  |  |
|  |  |
|  |  |
| Total Estimated Size: | 500 |

Code Review Checklist

Add comments. Boxes are checkable (+👆= ).

# Specification / Design

Is the functionality described in the specification fully implemented by the code?

Is there any excess functionality in the code but not described in the specification?

# Initialization and Declarations

Are all local and global variables initialized before use?

Are variables and class members of the correct type and appropriate mode

Are variables declared in the proper scope?

Is a constructor called when a new object is desired?

Are all needed import statements included?

Names are simple and if possible short

Comment: where the Names can be short they are kept short and easy to remember what they are doing.

There are no usages of “[magic numbers](https://stackoverflow.com/q/47882/5432315)”

# General

Code is easy to understand

Comment: The names are memeorable and the comments provide and easy way to see what is going on.

Variable and method names are spelt correctly

There is no dead code (i.e., code inaccessible at runtime)

Code is not repeated or duplicated

No empty blocks of code

Comment: yes there is no empty lines of code

# Method Calls

Are parameters presented in the correct order?

Comment: yes parameters appear in the right order and cannot be reverse, if so the code corrects the error by telling the user to write the apprioraite format

Are parameters of the proper type for the method being called?

Is the correct method being called, or should it be a different method with a similar name?

Are method return values used properly? Are they being cast to the needed type?

Comment: Methods return appropriate types and there is no casting necessary

# Arrays/Data structures

Are there any [off-by-one errors](https://stackoverflow.com/q/2939869/5432315) in array indexing?

comment: No there is no off by one error as they are handled if there were and correct by the code.

Can array indexes ever go out-of-bounds?

Comment: No they should not go out of bound but if they do the code has a try block to catch this exception and correct it.

Is a constructor called when a new array item is desired?

Are your data structures ideal?

Collections are initialized with a specific estimated capacity

# Object

Are all objects (including Strings) compared with equals and not ==?

Comment: the appropriate operator was used in the case.

No object exists longer than necessary

Comment: I make sure to teardown the object by calling it and setting it equal to null.

Files/Sockets and other resources if used are properly closed even if an exception occurs when using them

Comment: None were used, but if they were used they would have been handled properly.

# Output Format

Are there any spelling or grammatical errors in the displayed output?

Comment: Spelling for the error messages and anything Displayed to the user were double checked and spelled properly.

Is the output formatted correctly and consistently in terms of line stepping and spacing?

# Computation, Comparisons and Assignments

Check order of

computation/evaluation

operator precedence and

parenthesizing

Can the denominator of any divisions ever be zero?

Is integer arithmetic, especially division, ever used inappropriately, causing unexpected truncation/rounding?

Check each condition to be sure the proper [relational](https://docs.oracle.com/javase/tutorial/java/nutsandbolts/op2.html#PageContent:~:text=The%20Equality%20and%20Relational%20Operators,-The) and [conditional](https://docs.oracle.com/javase/tutorial/java/nutsandbolts/op2.html#PageContent:~:text=The%20Conditional%20Operators) operators are used.

If the test is an error-check, can the error condition be legitimate in some cases?

Does the code rely on any [implicit type conversions](https://docs.oracle.com/javase/specs/jls/se15/html/jls-5.html#logo:~:text=First%2C%20for%20some%20expressions%2C%20termed%20poly,appropriate%20type%2C%20a%20compile%2Dtime%20error%20occurs.)?

Comment: None were used, but if they were used they would have been handled properly.

# Exceptions

Are all relevant exceptions caught?

Comment: Exception caught were specific and made sense for where they were place as they were common in the code.

Is the appropriate action taken for each catch block?

comment: I make sure to notify the user about what has happened and return the appropriate value if exception is caught to stop the exception from breaking further parts of the program.

Are all appropriate exceptions thrown?

Are catch clauses fine-grained and catching specific exceptions?

# Flow of Control

In switch statements, is every case terminated by break or return?

Comment: All statements that were in a loop have a place where a termination Is reach or they return and no infinte loops exist

Do all switch statements have a default branch?

Check that nested if statements don't have “[dangling else](https://en.wikipedia.org/wiki/Dangling_else#:~:text=The%20dangling%20else%20is%20a,than%20one%20correct%20parse%20tree.)” problems.

Are all loops correctly formed, with the appropriate initialization, increment and termination expressions?

Are open-close parentheses and brace pairs properly situated and matched?

# Files

Are all files properly declared and opened?

Comment: None were used, but if they were used they would have been handled properly.

Are all files closed properly, even in the case of an error?

Comment: None were used, but if they were used they would have been handled properly.

Are [EOF](https://en.wikipedia.org/wiki/End-of-file#:~:text=In%20computing%2C%20end%2Dof%2D,called%20a%20file%20or%20stream.) conditions detected and handled correctly?

Are all file exceptions caught?

Comment: None were used, but if they were used they would have been handled properly.

# Documentation

Methods commented in clear language

Comment: I make sure to provide comments either above or in the method itself to describe what it is doing so there is no confusion at what my tests are doing.

Most comments should describe rationale or reasons (the *why*); fewer should describe the *what*; few should describe *how*.

Are there any out-of-date comments that no longer match their associated code?

All public methods/interfaces/contracts are commented describing usage

All edge cases are described in comments

Comment: In the J unit Testing I use comment to denote where I am testing any edge cases in my coe so it is explicitly clear where I am doing it

All unusual behavior or edge case handling is commented

Data structures and units of measurement are explained

PSP Time Recording Log

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Date | Start | Stop | Interruption Time | Delta  Time | Phase | Comments |
| 11/22 | 12:00 PM | 12:50 PM | - | 50 minutes | Planning | The requirements were taken into account and understood, and a plan of action was taken |
| 11/22 | 12:50 PM | 1:15PM | - | 25 minutes | Design | The requirements given where made into tangible designs that gave a big picture view of what we were testing |
| 11//22 | 2:00pm | 3:25pm | 5 minutes | 80 minutes | Code | Created the first part of the test cases for isValid |
| 11/22 | 3:40 pm | 4:40 pm | - | 60 minutes | Code | Created the second part for testing isValid |
| 11/22 | 5:30pm | 6:00pm | - | 30 minutes | Test | Tested Junit test cases |
| 11/22 | 6:30pm | 8:30pm | - | 120 minutes | Coding | Additional cases and methods had to be considered for testing |
| 11/22 | 10:50 pm | 11:20pm | - | 30 mintues | Coding | Working on Computer Player Test Case |
| 11/23 | 10:00 am | 10:30 am | - | 30 minutes | Coding | Work on Computer player and finish it |
| 11/23 | 10:35am | 11:20 am | 5 | 40 minutes | Coding/testing | Cover more tests with the logic class |
| 11/23 | 11:20 am | 11:30 am | - | 10 minutes | Review | Went through list and check what was done and what needed to be done |
| 11/23 | 11:30 am | 12:00pm | - | 30 minutes | Postmortem | Questions and table |
|  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |

* **Interruption time**: Record any interruption time that was not spent on the task. Write the reason for the interruption in the "Comment" column. If you have several interruptions, record them with plus signs (to remind you to total them).
* **Delta Time**: Enter the clock time you spent on the task, less the interrupt time.
* **Phase**: Enter the name or other designation of the programming phase being worked on. Example: Design or Code.
* **Comments**: Enter any other pertinent comments that might later remind you of any details or specifics regarding this activity.

PSP Defect Recording Log

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| Serial No. | Date | Defect Type No. | Defect Inject Phase | Defect Removal Phase | Fix Time (duration) | Fix Ref | Description |
| 1 | 11/22 | 20 | code | code | 10 | X | The spelling one of my variables was messed up and had to be fixed |
| 2 | 11/22 | 100 | code | test | 20 | X | I didn’t have the Junit library in setup and had to set it up |
| 3 | 11/23 | 60 | design | code | 10 | X | To achieve further code coverage, I had to provide more checks |
| 4 | 11/23 | 10 | code | review | 5 | X | Detailed comments were written |
|  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |

* **Serial No.**: The unique id you associate with the defect; allows you to reference it later.
* **Defect Type No.**: The type number of the type—see the PSP Defect Type Standard table below and use your best judgement.
* **Defect Inject Phase**: Enter the phase (plan, design, code, etc.) when this defect was injected using your best judgment.
* **Defect Removal Phase**: Enter the phase during which you fixed the defect.
* **Fix Time**: Enter the amount of time that you took to find and fix the defect.
* **Fix Ref**: If you or someone else injected this defect while fixing another defect, record the number of the improperly fixed defect. If you cannot identify the defect number, enter an X. If it is not related to any other defect, enter N/A.
* **Description**: Write a succinct description of the defect that is clear enough to later remind you about the error and help you to remember why you made it.

**PSP Defect Type Standard**

|  |  |  |
| --- | --- | --- |
| Type Number | Type Name | Description |
| 10 | Documentation | Comments, messages |
| 20 | Syntax | Spelling, punctuation, typos, instruction formats |
| 30 | Build, Package | Change management, library, version control |
| 40 | Assignment | Declaration, duplicate names, scope, limits |
| 50 | Interface | Procedure calls and references, I/O, user formats |
| 60 | Checking | Error messages, inadequate checks |
| 70 | Data | Structure, content |
| 80 | Function | Logic, loops, recursion, computation, function defects |
| 90 | System | Configuration, timing, memory |
| 100 | Environment | Design, compile, test, or other support system problems |

PSP2 Project Summary

**Time in Phase**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Phase | Estimated time (in minutes) | Actual time (in minutes) | To Date | % of total time to Date |
| Planning | 40 | 50 | 145 | 5% |
| Design | 50 | 25 | 124 | 4% |
| Code | 200 | 340 | 1599 | 55% |
| Code Review | 30 | 10 | 305 | 11% |
| Test | 100 | 50 | 529 | 18% |
| Postmortem | 30 | 30 | 190 | 7% |
| TOTAL | 450 | 505 | 2892 | 100% |

**Defects Injected**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Phase | Estimated Defects | Actual Defects | To Date | % of total to Date |
| Planning | 0 | 0 | 2 | 12% |
| Design | 1 | 1 | 3 | 18% |
| Code | 2 | 3 | 12 | 70% |
| Code Review | 1 | 0 | 0 | 0 |
| Test | 2 | 0 | 0 | 0 |
| Postmortem | 0 | 0 | 0 | 0 |
| TOTAL | 6 | 4 | 17 | 100% |

**Final Summary**

|  |  |  |  |
| --- | --- | --- | --- |
| Metric | Estimated | Actual | To Date |
| Program Size (Lines of Code—LOC) [[1]](#footnote-1) | 400 | 525 | 2071 |
| Productivity (calculated by LOC/Hour) | 50 | 62 | 0.72 |
| Defect Rate (calculated by Defects/KLOC) [[2]](#footnote-2) | 12 | 7.6 | 8.2 |

## Reflection Questions

1. How good was your time *and defect* estimate for various phases of software development?

I think my Time and defects are getting much better because for the time estimations I was able to get close with my estimated and actual planning times. I was spot on with my Postmortem time. I was a fair bit off with my code and code review time. I noticed that I spent a lot more time coding than what I though and I spent less time code revieing then I thought. Overall, my estimations appear to be getting much closer to their actual values. My defect estimation was off by 2 but that is understandable since I spent more time on planning then have every spent before. I was able to eliminate a few of the major defect I would have received had I spent less time.

1. How good was your program size estimate, i.e., was it close to actual?

My program size estimation was less than 120 lines off from my actual. This is much better than my previous estimations. I also could see why I might have written more code because as I ran through code coverage I came across instances where I failed to check something.

1. How much code coverage did you achieve for your core classes?

In Total I achieve close to 75% code coverage for my core class all together. The reason why was due to a lot of extra returns that would never be reach that needed to be fix. My test cases went over a healthy amount of input that it might have receive and I feel that this is a strong bit of code coverage for lines. I was able to achieve 100% method coverage for my core class which mean I used every method as I should.

PSP Design Form

Use this form to record whatever you do during the design phase of development. Include notes, class diagrams, flowcharts, formal design notation, or anything else you consider to be part of designing a solution that happens BEFORE you write program source code. Attach additional pages if necessary.

*A picture containing diagram

Description automatically generated*

Possible Tests for testIsValid:

* a piece off the bord
* a piece that is on the board that can move
* a piece on the edge of the board that can and cannot move
* a piece that is all te way at the end

Possible Tests for Test Getboard:

* make a capture and then get the board
* don’t make a move

Possible capture tests

* try capturing the same piece
* try capturing opposing piece
* try capturing empty space

Possible tests for to test move

* see if the computer can move in a spot outside the board
* while they are stuck
* at the end

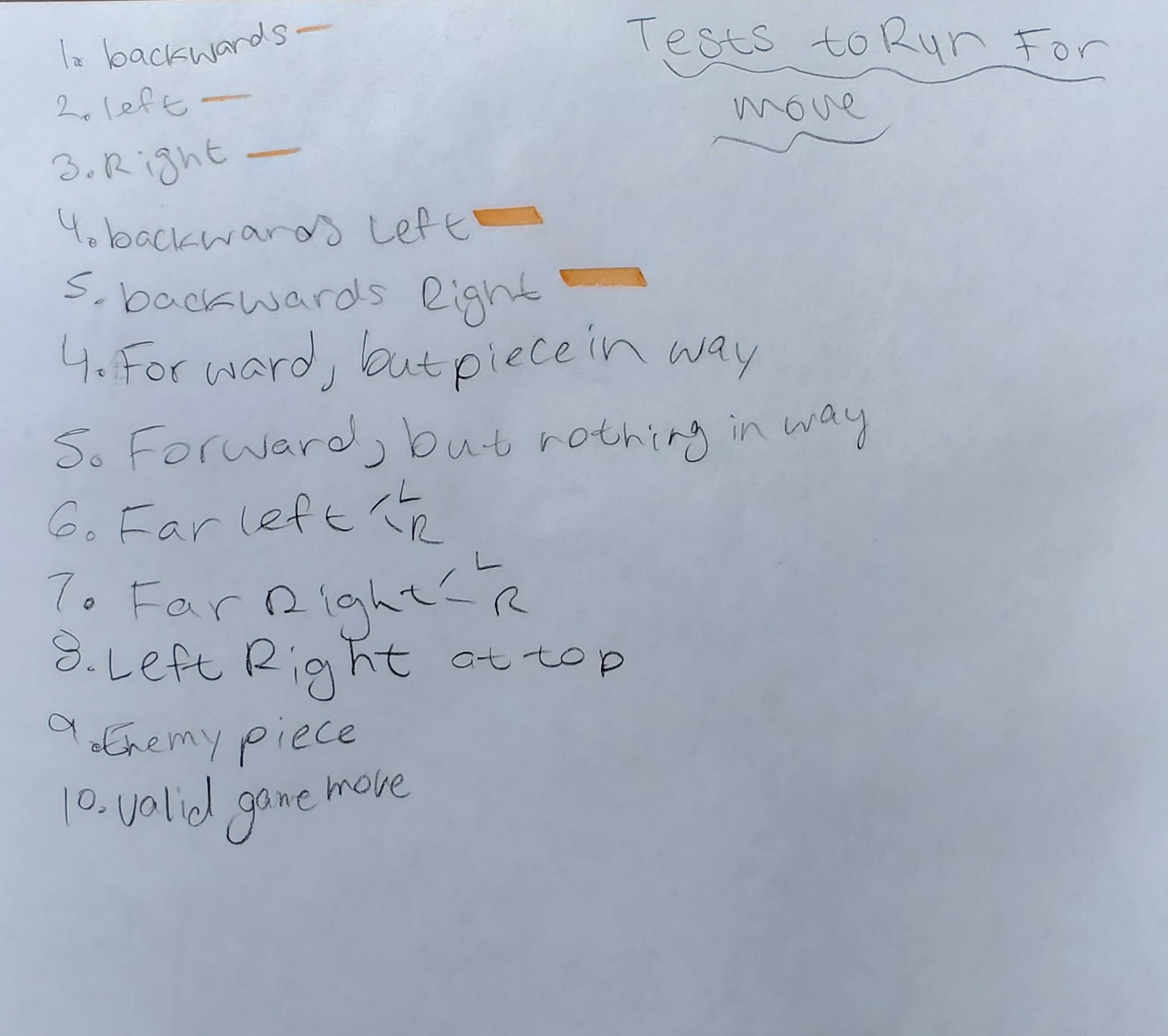
Possible move left

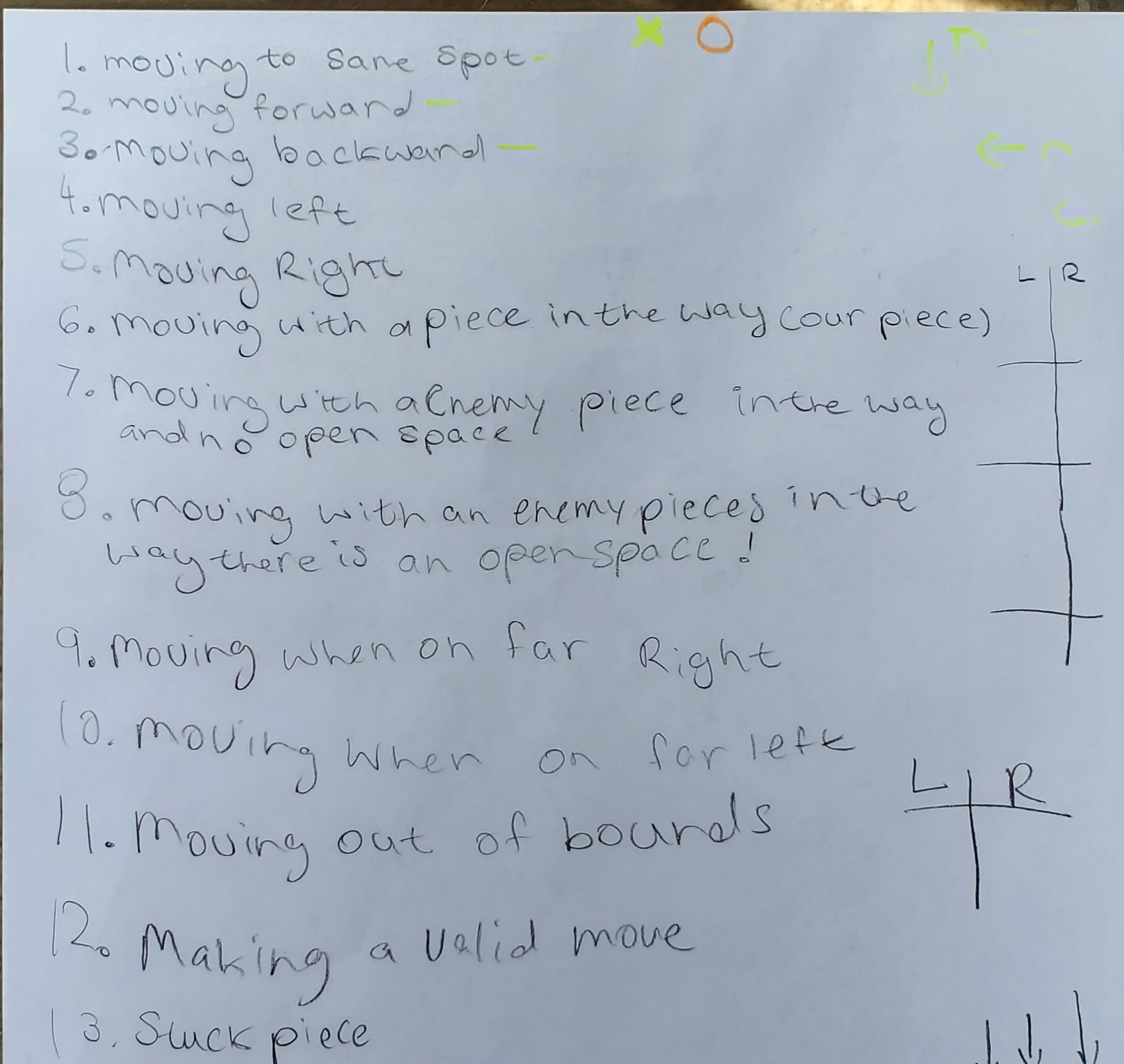
* move left if there if moving left goes outside the board
* move left in the board

Possible move right

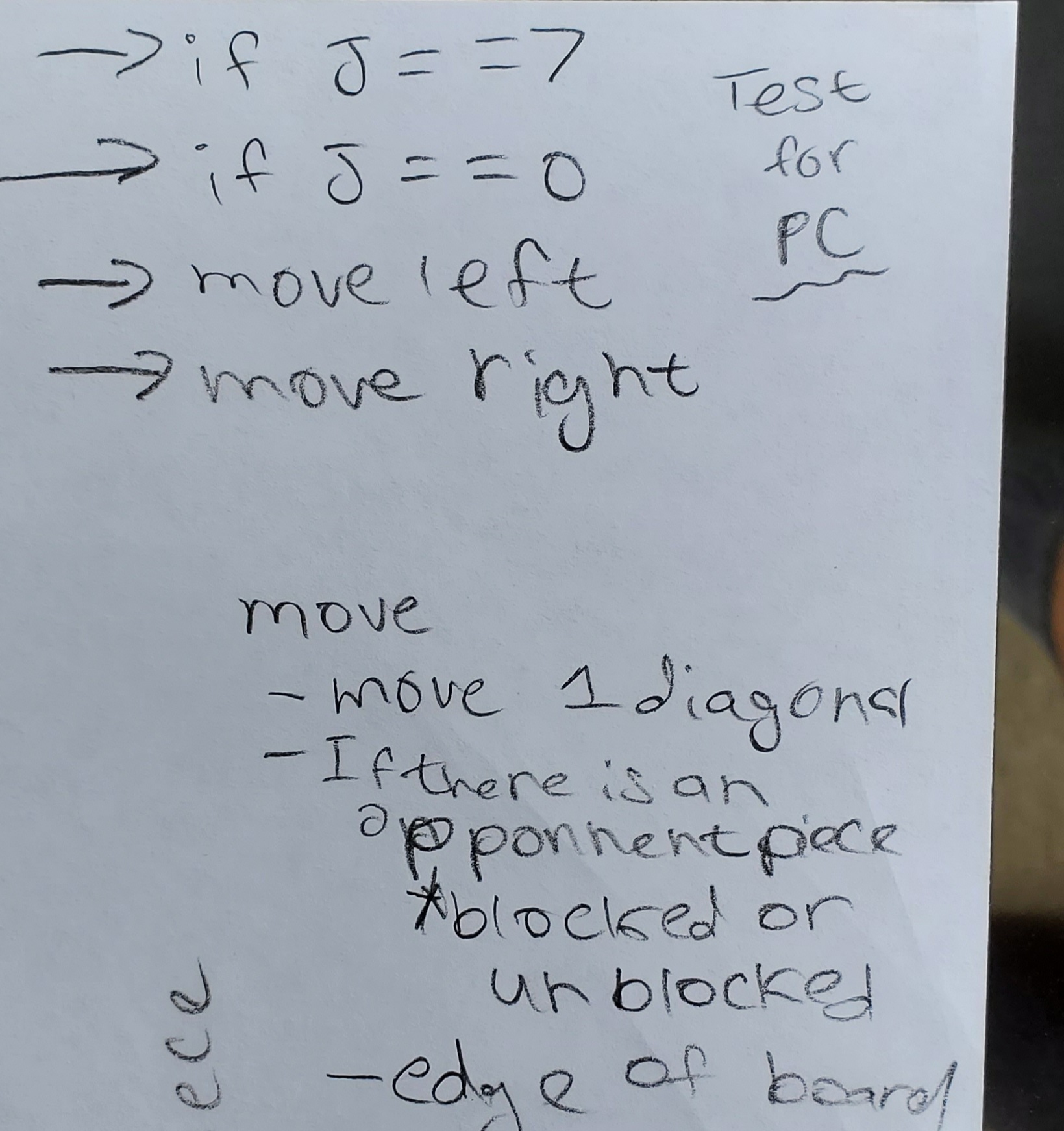
* move right if that takes us outside the board
* move right in the board

Possible Test Cases Plan for Move





Test to run on computer player



1. LOC stands for lines of code. [↑](#footnote-ref-1)
2. KLOC stands for kilo lines of code (1000 lines) [↑](#footnote-ref-2)