SE 576 Programming Assignment
Chess Validation Engine Report

Victoria Thacker

Contents

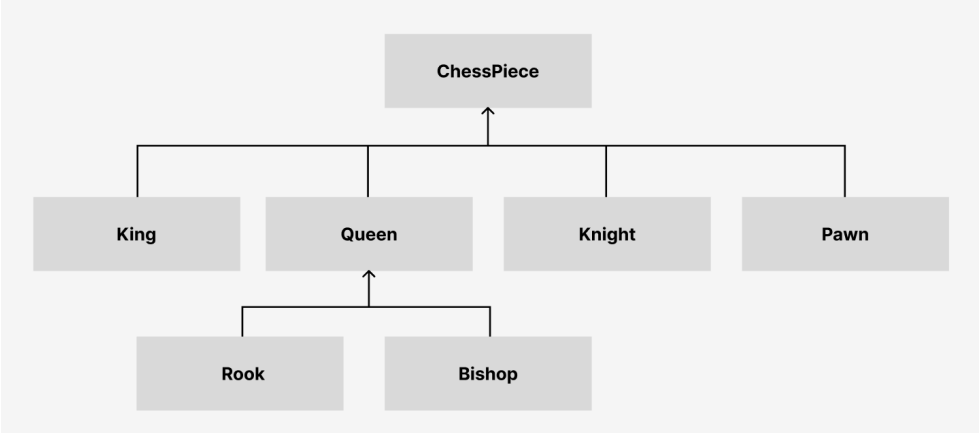
1 Design of the Software System

The Chess Validation Engine was developed and tested in Java, using the Intellij IDEA IDE. I approached this project by creating classes for each chess piece and organizing them into an inheritance hierarchy based on their behavior, while having other classes outside of the hierarchy assist in functionality.

The final version of the software system consists of nine classes:

1. **ChessDriver**: the driver class which accepts user input, creates appropriate objects based on user input, and outputs the possible moves for a given chess piece
2. **ChessPiece**: the parent class of all of the functional chess pieces
3. **King**: defines the behavior of a King chess piece
4. **Queen**: defines the behavior of a Queen chess piece and is the parent class of Bishop and Rook
5. **Bishop**: defines the behavior of a Bishop chess piece by calling applicable methods from Queen
6. **Rook**: defines the behavior of a Rook chess piece by calling applicable methods from Queen
7. **Knight**: defines the behavior of a Knight chess piece
8. **Pawn**: defines the behavior of a Pawn chess piece
9. **ScannerWrapper**: a singleton wrapper class for a Scanner object

The inheritance hierarchy of the functional classes is shown below:



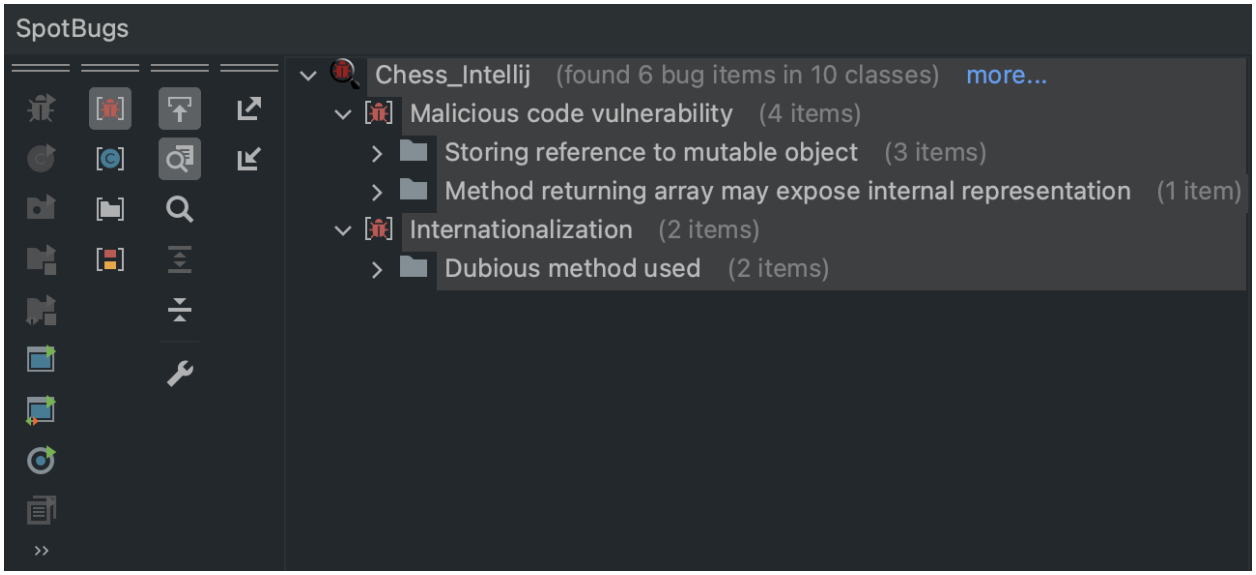
Prior to static analysis and testing, two additional classes were present: Board and ColorPieces. Board was responsible for printing a chess board and was used for testing the system as it was being constructed, but it provided no functionality outside of this. ColorPieces was created early in the development of the system but was no longer needed as the system evolved. Both of these classes were removed in the final version of the system, but are mentioned throughout this report due to their inclusion in the static analysis. ScannerWrapper was added during the static analysis.

2 Static Analysis

The SpotBugs static analysis tool was used for the static analysis portion of the assignment, which is the successor to the University of Maryland's FindBugs static analysis tool. SpotBugs was chosen because I was familiar with FindBugs from class and because SpotBugs has an Intellij plugin to easily use the software inside the IDE.

When running SpotBugs for the first time, it was run using the "Analyze Project Files Not Including Test Sources" option. The project files included all of the classes listed in the Design portion of this report except ScannerWrapper, and also included the Board and ColorPieces classes. No tests were created before static analysis was conducted.

The initial SpotBugs report is shown below:



SpotBugs found six bug items in the code: four categorized as "malicious code vulnerability" and two categorized as "internalization". The malicious code vulnerability bugs were all indicated as medium priority and the internalization bugs were indicated as high priority by SpotBugs. Of the malicious code vulnerability bugs, three were due to "storing reference to a mutable object", each in the ChessPiece class, and one was due to a "method returning an array" in the ColorPieces class, which the SpotBugs documentation stated may cause the program to be exposed to internal representation by returning a reference to a mutable object. Both internalization bugs were due to the instantiation of the Scanner class in the driver class, which was used to accept user input in two different methods.

The internalization bugs were addressed first because they were both caused by the same issue and they were marked as high priority. The warning for both of these bugs stated:

> Found a call to a method which will perform a byte to String (or String to byte) conversion, and will assume that the default platform encoding is suitable. This will cause

the application behavior to vary between platforms. Use an alternative API and specify a charset name or Charset object explicitly.

Below is a screenshot of the initial code in the ChessDriver class that caused SpotBugs to issue this warning:

```
57 @    public static String[] getBoardInput(String color) {
58          Scanner scan = new Scanner(System.in);
59          System.out.print(color + ": ");
60          String input = scan.nextLine();
61          // Putting current pieces into arrays
62          return input.split( regex: ", ");
63      }
64
65      // Gets input for the piece that the user wants to move
        1 usage
66      public static String getPieceToMove() {
67          Scanner scan = new Scanner(System.in);
68          System.out.print("PIECE TO MOVE: ");
69          return scan.nextLine();
70      }
```

After researching the content of the bug warning, I did not have an immediate fix to this bug because I was not familiar with specifying charsets in Java. Because there were two Scanner objects, I decided to start addressing this by creating a singleton wrapper class for Scanner. The ScannerWrapper class was created, which contains a nextLine() method that was used in the user input methods in the driver class:

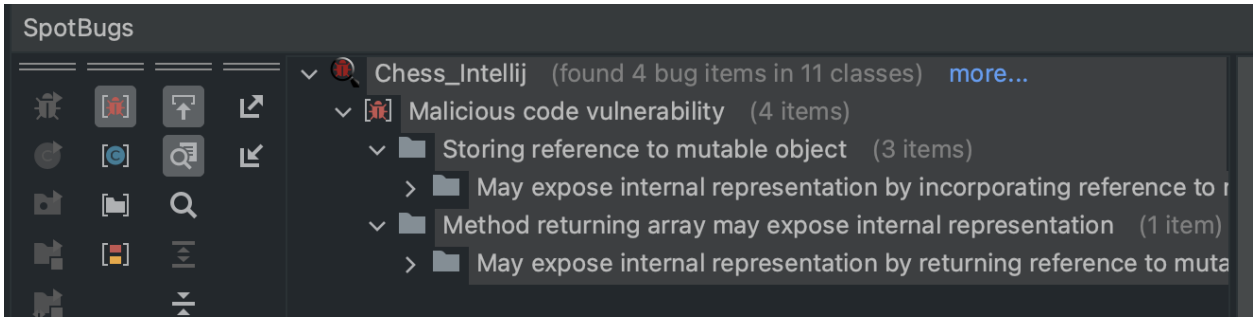
```
public String[] getBoardInput(ScannerWrapper scannerWrapper, String color) {
    System.out.print(color + ": ");
    String input = scannerWrapper.nextLine();
    // Putting current pieces into arrays
    return input.split( regex: ", ");
}

// Gets input for the piece that the user wants to move
1 usage
public String getPieceToMove(ScannerWrapper scannerWrapper) {
    System.out.print("PIECE TO MOVE: ");
    return scannerWrapper.nextLine();
}
```

After running SpotBugs a second time, the report showed five bugs instead of six. However, the same error was identified because a charset was still not identified. Although the ScannerWrapper class did not fix the bug, it was kept in the system to adhere to good programming practices and to only have to fix the internalization error once.

After more research, I found that UTF-8 is the default platform encoding mechanism on most operating systems including Windows OS, Mac OS, and Linux. I found that the Scanner class can accept a platform encoding mechanism as a constructor argument, so the ScannerWrapper class was edited to account for UTF-8:

```
1 usage
private ScannerWrapper() {
    scanner = new Scanner(System.in, StandardCharsets.UTF_8);
}
```

After adjusting the Scanner constructor arguments, the third run of SpotBugs indicated that the internalization bug was resolved. Now, SpotBugs reported only four bugs in the system, which were the remaining malicious code vulnerability bugs:



All four of these bugs related to the ColorPieces class. The “method returning array [that] may expose internal representation” is located in the ColorPieces class and is called by the driver class to create a String array, which is used later in the driver class. The “storing reference to mutable object” bugs refer to the passing of these String arrays to the constructor of ChessPiece, the parent class for the functional pieces.

Below is the bug in the getPiecePosition() method of the ColorPieces class:

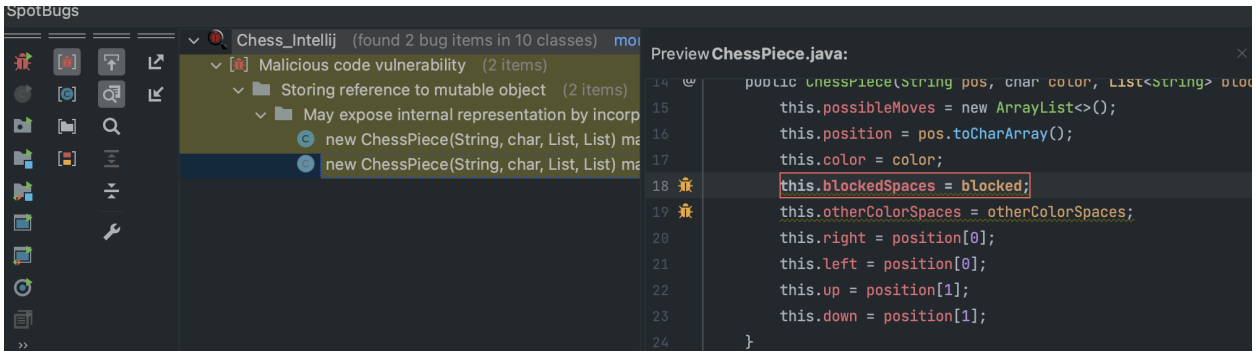
```
12      public String[] getPiecePosition() {
13          for (int i = 0; i < pieces.length; i++) {
14              this.positions[i] = pieces[i].substring(1);
15          }
16          return positions;
17      }
```

Below are the bugs in the constructor of ChessPiece (note that Lists blockedSpaces and otherColorSpaces are initialized via the getPiecePosition() method of the ColorPiece class and converted to Lists in the driver class):

```
    4 usages
14  public ChessPiece(String pos, char color, List<String> blocked, List<String> otherColorSpaces) {
15      this.possibleMoves = new ArrayList<>();
16      this.position = pos.toCharArray();
17      this.color = color;
18      this.blockedSpaces = blocked;
19      this.otherColorSpaces = otherColorSpaces;
20      this.right = position[0];
21      this.left = position[0];
22      this.up = position[1];
23      this.down = position[1];
24  }
```

The functionality of the ColorPieces class was created early in my development of the system, but because of how the system has evolved, it no longer needed to be a class. I began to address these bugs by removing the ColorPieces class entirely, placing the getPiecePosition() method in the driver class, and adjusting the method calls in the driver class to account for those changes.

SpotBugs was run a fourth time with the following results:

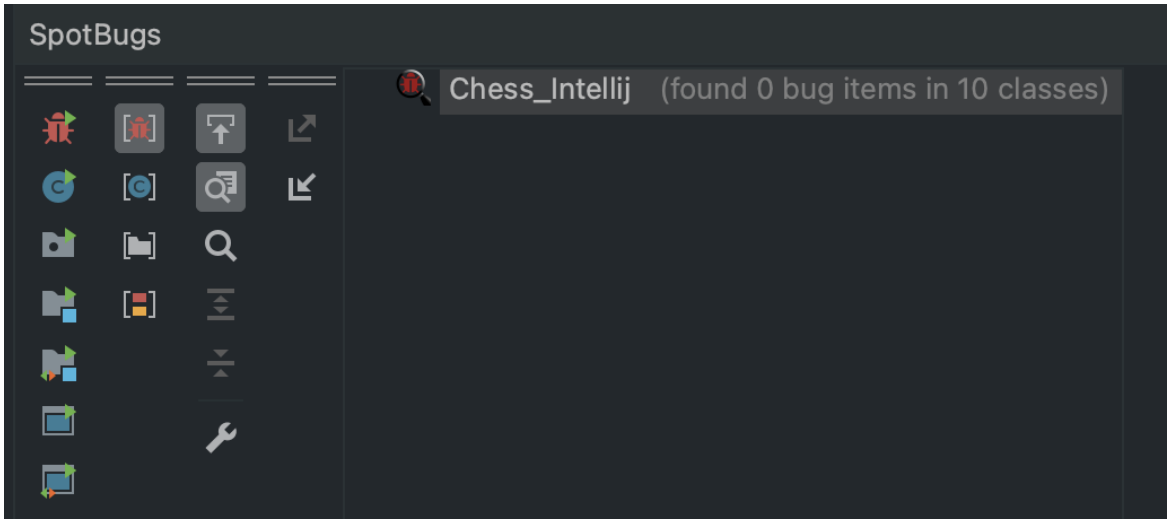


SpotBugs still reported malicious code vulnerability bugs, but only two remained - both in the constructor of ChessPiece. In the SpotBugs documentation for this error, it suggested that a copy of the object should be used rather than the object itself.

To address this, I used the copyOf() method of the List class to assign a copy of the Lists from the constructor argument to the Lists in the class:

```
public ChessPiece(String pos, char color, List<String> blocked, List<String> otherColorSpaces) {
    this.possibleMoves = new ArrayList<>();
    this.position = pos.toCharArray();
    this.color = color;
    this.blockedSpaces = List.copyOf(blocked);
    this.otherColorSpaces = List.copyOf(otherColorSpaces);
    this.right = position[0];
    this.left = position[0];
    this.up = position[1];
    this.down = position[1];
}
```

After that change, the fifth run of SpotBugs yielded no bugs in the system.



After creating tests for the software system, I removed the Board class because it provided no functionality. I ran SpotBugs a final time and received the same results, but the message accounted for nine classes instead of 10.

3 Testing

The requirements of the Chess Validation Engine state that:

1. The program will compute all legal moves for a piece on a given chess board configuration.
    a. Legal moves are defined by the Rules of Chess Wikipedia article.
2. The program should accept as input a board configuration that contains the position of white pieces, the position of black pieces, and the piece whose possible legal moves are to be computed as such:

    WHITE: Rf1, Kg1, Pf2, Ph2, Pg3
    BLACK: Kb8, Ne8, Pa7, Pb7, Pc7, Ra5
    PIECE TO MOVE: Rf1
3. Given the input description, the program should produce output in the following format:

    LEGAL MOVES FOR Rf1: e1, d1, c1, b1, a1

A series of tests were created to assess the reliability of the program based on these requirements and were adjusted after running code coverage analyses to ensure that the program was as thoroughly tested as possible. The tests consist of unit tests and system tests. Specific integration tests were not included because the simplicity of the inheritance hierarchy and ubiquitousness of the driver class allow integration testing to occur within the unit and system tests.

3.1 Unit Tests

JUnit white box tests were developed to assess the correctness of the functionality of the software system. The development of these tests began with attention to the system's adherence to its requirements, and then were expanded to increase code coverage. There are 59 JUnit white box tests in addition to 10 system tests included in the test zip folder.

There are several conditions that must be met in order for the tests to work as intended. These conditions are outlined in detail in the readme.md file included with the tests, but below is a brief overview:

Test setup requirements:

1. JUnit 5 must be imported before the tests are run.
2. Mockito must be imported via Maven before the tests are run.

Input requirements:

1. Input must match the format prescribed in the requirements exactly.
2. The PIECE TO MOVE must exist within the input string of either the white or black pieces.

3.2 System Tests

The following system tests examine the functionality of the system as a whole via a black box testing method. One board configuration was used in all of these tests, and each piece highlighted in the board configuration was tested individually for:

1. Inclusion of all legal moves in the output, including allowable captures of an opponent's pieces.

2. Maintaining the bounds of the chess board (i.e. a chess piece may not move to square a9 or i5, etc.).

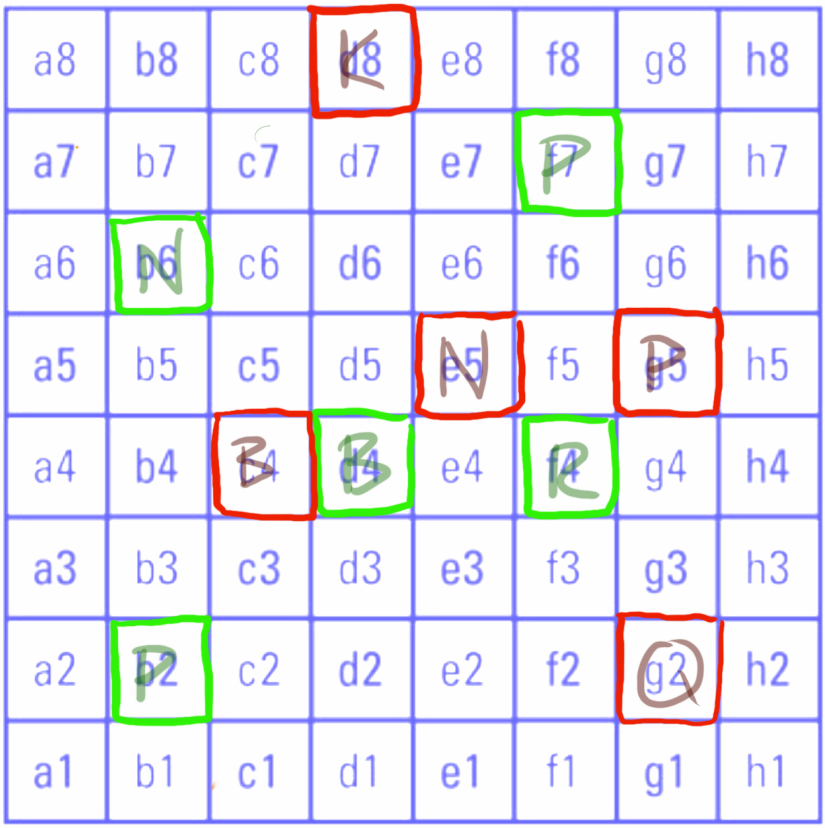
Expected possible moves for each test are shown by yellow circles on the board and are compared to output from the software system. Note that in the given images, green squares denote white pieces and red squares denote black squares for clarity. Output may not be in the same order as the expected output, but output is correct if it includes all of the expected moves and no additional moves. All six types of chess pieces are represented in this board configuration. Only pawn functionality differs between white and black pieces, so pawns were included in both the white and black input.

These tests are also represented as JUnit tests in the ChessDriverTest testing class.

The image below shows a chess board configuration with the following inputs:

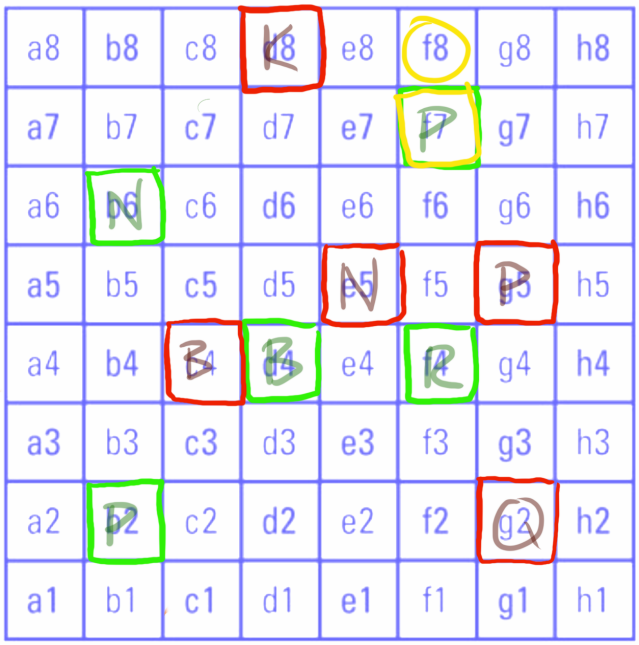
WHITE: Pf7, Nb6, Bd4, Rf4, Pb2
BLACK: Kd8, Bc4, Ne5, Pg5, Qg2

This board configuration is used for all of the tests, and each piece on the board was manipulated individually during its respective test. K denotes a king, Q denotes a queen, B denotes a bishop, R denotes a rook, N denotes a knight, and P denotes a pawn.



3.2.1 Test 1

The first PIECE TO MOVE is Pf7, a white pawn. Pawns move one space forward, except if they are located in their starting position before the move is performed. They can also capture one space diagonally forward in either direction. Because the white pawn at Pf7 is neither in its starting place nor are there any allowable captures, the only allowable move for this piece is [f8].



Input for this test case was:

WHITE: Pf7, Nb6, Bd4, Rf4, Pb2
BLACK: Kd8, Bc4, Ne5, Pg5, Qg2
PIECE TO MOVE: Pf7

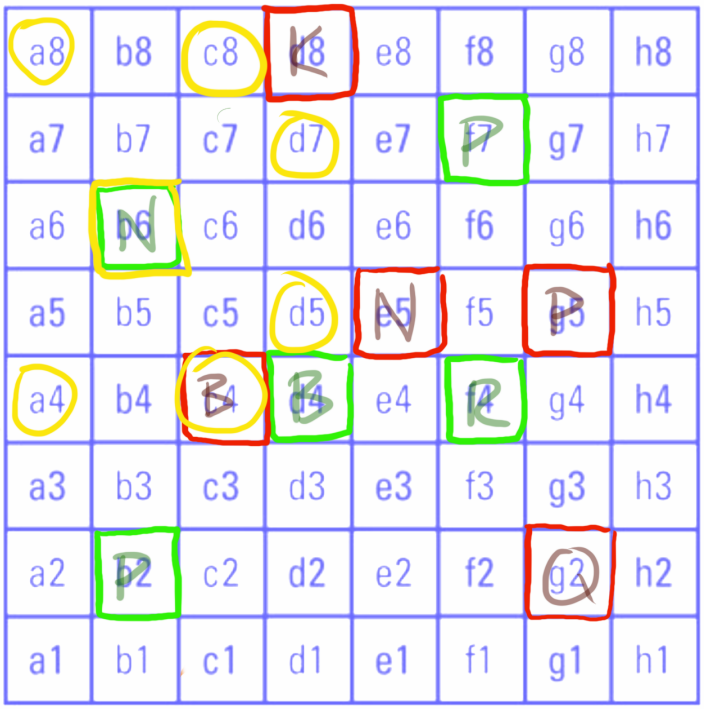
Program output was:

```
WHITE: Pf7, Nb6, Bd4, Rf4, Pb2
BLACK: Kd8, Bc4, Ne5, Pg5, Qg2
PIECE TO MOVE: Pf7
LEGAL MOVES FOR WHITE Pf7: [f8]
```

Result: the output of the program matched the expected output of Test 1.

3.2.2 Test 2

The next PIECE TO MOVE is Nb6, which is a white knight. Knights can move forward two spaces and sideways one space in any direction, or move forward one space and sideways two spaces in any direction. Knights, like all pieces, may not move off the board. They may capture an opposing piece if the piece occupies a square in the allowable move range. The image below shows that the possible moves for the white knight at square c4, including a possible capture of a black piece in square d3, are [a8, c8, d7, d5, c4, a4].



Input for this test case was:

WHITE: Pf7, Nb6, Bd4, Rf4, Pb2
BLACK: Kd8, Bc4, Ne5, Pg5, Qg2
PIECE TO MOVE: Nb6

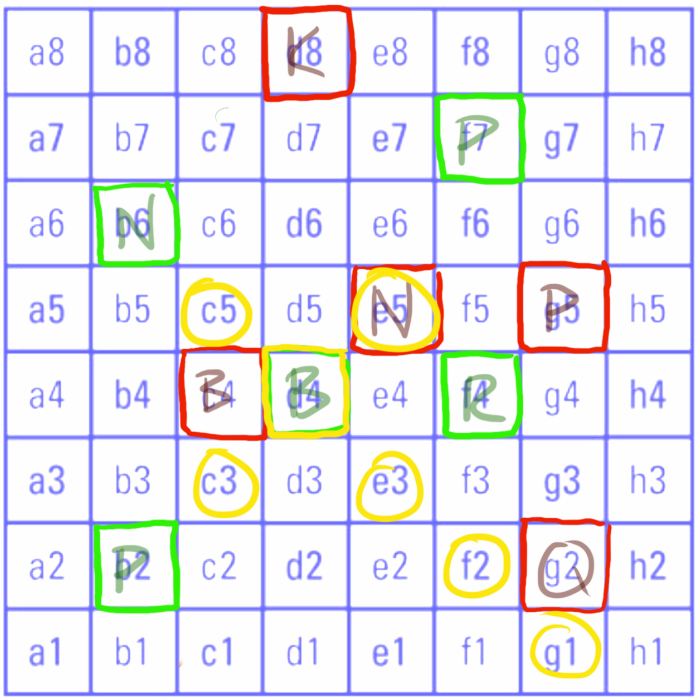
Program output was:

```
WHITE: Pf7, Nb6, Bd4, Rf4, Pb2
BLACK: Kd8, Bc4, Ne5, Pg5, Qg2
PIECE TO MOVE: Nb6
LEGAL MOVES FOR WHITE Nb6: [d7, c8, a8, d5, c4, a4]
```

Result: the output of the program matched the expected output of Test 2.

3.2.3 Test 3

The next PIECE TO MOVE is Bd4, a white bishop. Bishops can move diagonally in any direction until they reach the edge of the board or until they encounter another piece. The image below shows that the possible moves for a white bishop at square d4, including a possible capture, include [c5, e5, c3, e3, f2, g1].



Input for this test case was:

WHITE: Pf7, Nb6, Bd4, Rf4, Pb2
BLACK: Kd8, Bc4, Ne5, Pg5, Qg2
PIECE TO MOVE: Bd4

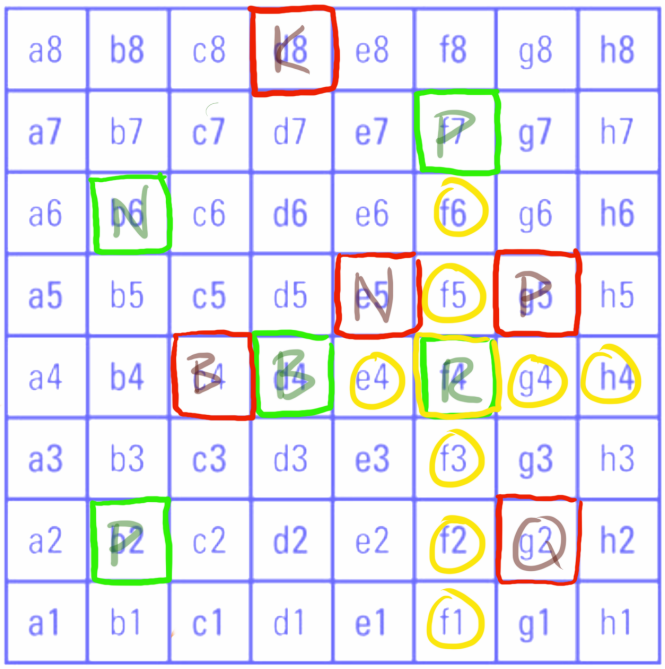
Program output was:

```
WHITE: Pf7, Nb6, Bd4, Rf4, Pb2
BLACK: Kd8, Bc4, Ne5, Pg5, Qg2
PIECE TO MOVE: Bd4
LEGAL MOVES FOR WHITE Bd4: [e5, e3, f2, g1, c5, c3]
```

Result: the output of the program matched the expected output of Test 3.

3.2.4 Test 4

The next PIECE TO MOVE is Rf4, a white rook. Rooks can move forward, backward, left, and right until they reach the edge of the board or until they encounter another piece. The image below shows that the possible moves for a white rook at square f4 include [e4, f5, f6, g4, h4, f3, f2, f1].



Input for this test case was:

WHITE: Pf7, Nb6, Bd4, Rf4, Pb2
BLACK: Kd8, Bc4, Ne5, Pg5, Qg2
PIECE TO MOVE: Rf4

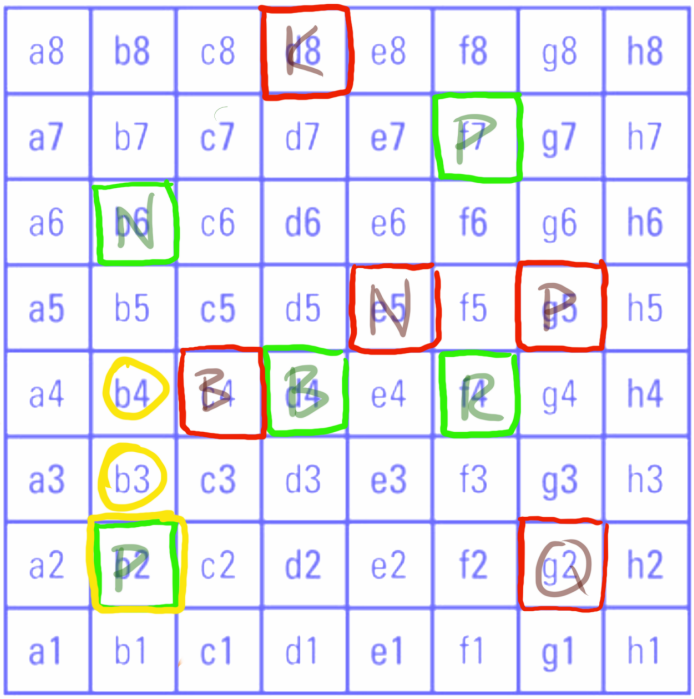
Program output was:

```
WHITE: Pf7, Nb6, Bd4, Rf4, Pb2
BLACK: Kd8, Bc4, Ne5, Pg5, Qg2
PIECE TO MOVE: Rf4
LEGAL MOVES FOR WHITE Rf4: [f5, f6, f3, f2, f1, e4, g4, h4]
```

Result: the output of the program matched the expected output of Test 4.

3.2.5 Test 5

The last white PIECE TO MOVE is Pb2, which is a white pawn. Because this pawn is located in its starting place, it may move either one or two squares forward. There are no allowable captures in this case and pawns are not permitted to move backwards. The image below shows that the possible moves for the white pawn located at b2 should be [b3, b4].



Input for this test case was:

WHITE: Pf7, Nb6, Bd4, Rf4, Pb2
BLACK: Kd8, Bc4, Ne5, Pg5, Qg2
PIECE TO MOVE: Pb2

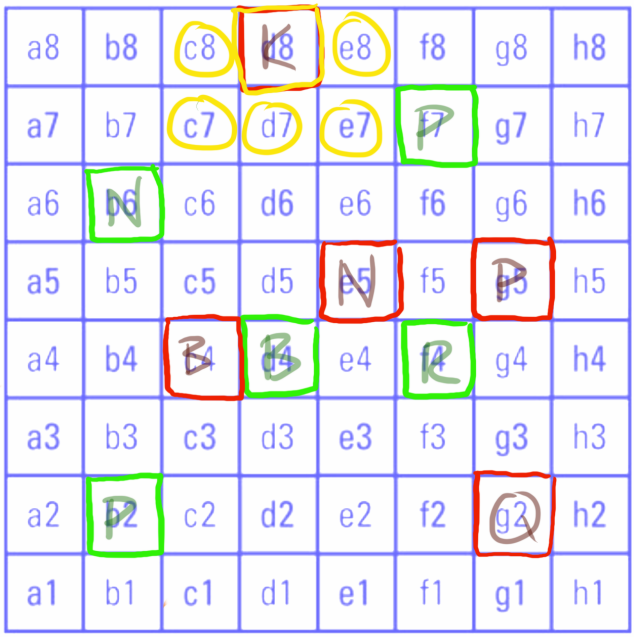
Program output was:

```
WHITE: Pf7, Nb6, Bd4, Rf4, Pb2
BLACK: Kd8, Bc4, Ne5, Pg5, Qg2
PIECE TO MOVE: Pb2
LEGAL MOVES FOR WHITE Pb2: [b3, b4]
```

Result: the output of the program matched the expected output of Test 5.

3.2.6 Test 6

The first black PIECE TO MOVE is Kd8, a king. Kings can move forward, backward, right, left, and diagonally in either direction, but they may only move one space. The image below shows that the black king at square d8 may move to [c8, c7, d7, e7, e8].



Input for this test case was:

WHITE: Pf7, Nb6, Bd4, Rf4, Pb2
BLACK: Kd8, Bc4, Ne5, Pg5, Qg2
PIECE TO MOVE: Kd8

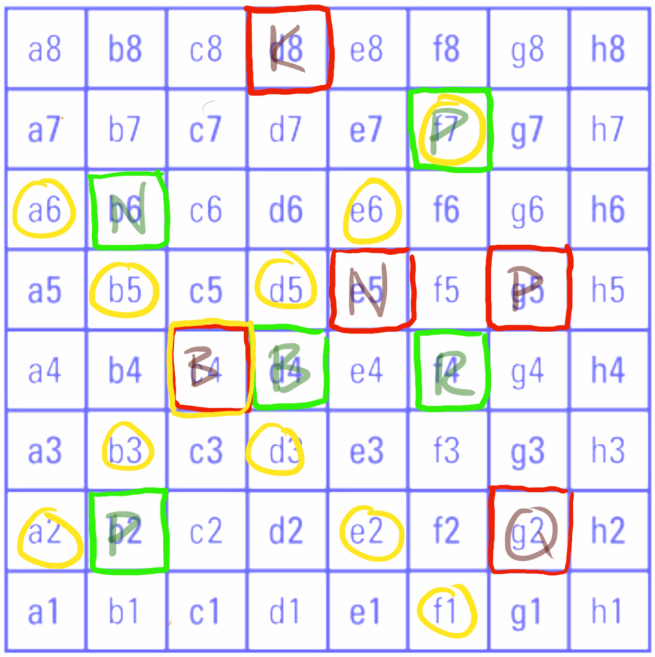
Program output was:

```
WHITE: Pf7, Nb6, Bd4, Rf4, Pb2
BLACK: Kd8, Bc4, Ne5, Pg5, Qg2
PIECE TO MOVE: Kd8
LEGAL MOVES FOR BLACK Kd8: [c8, e8, d7, c7, e7]
```

Result: the output of the program matched the expected output of Test 6.

3.2.7 Test 7

The next PIECE TO MOVE is Bc4, a black bishop. The allowable moves are shown in the image below and include [b5, a6, d5, e6, f7, b3, a2, d3, e2, f1].



Input for this test case was:

WHITE: Pf7, Nb6, Bd4, Rf4, Pb2
BLACK: Kd8, Bc4, Ne5, Pg5, Qg2
PIECE TO MOVE: Bc4

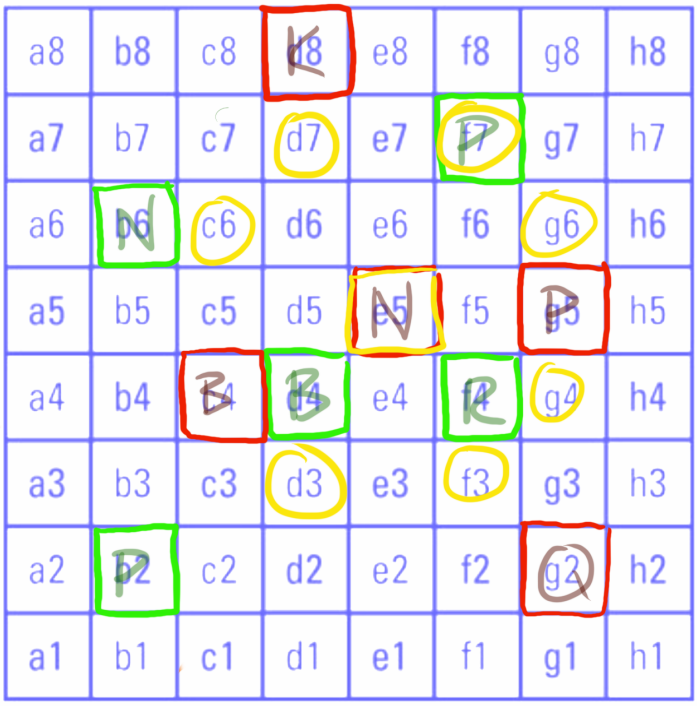
Program output was:

```
WHITE: Pf7, Nb6, Bd4, Rf4, Pb2
BLACK: Kd8, Bc4, Ne5, Pg5, Qg2
PIECE TO MOVE: Bc4
LEGAL MOVES FOR BLACK Bc4: [d5, e6, f7, d3, e2, f1, b5, a6, b3, a2]
```

Result: program output matched the expected output of Test 7.

3.2.8 Test 8

The next PIECE TO MOVE is Ne5, a black knight. The image below shows the allowable moves, including a capture of a white pawn, to be [c6, d7, f7, g6, g4, f3, d3].



Input for this test case was:

WHITE: Pf7, Nb6, Bd4, Rf4, Pb2
BLACK: Kd8, Bc4, Ne5, Pg5, Qg2
PIECE TO MOVE: Ne5

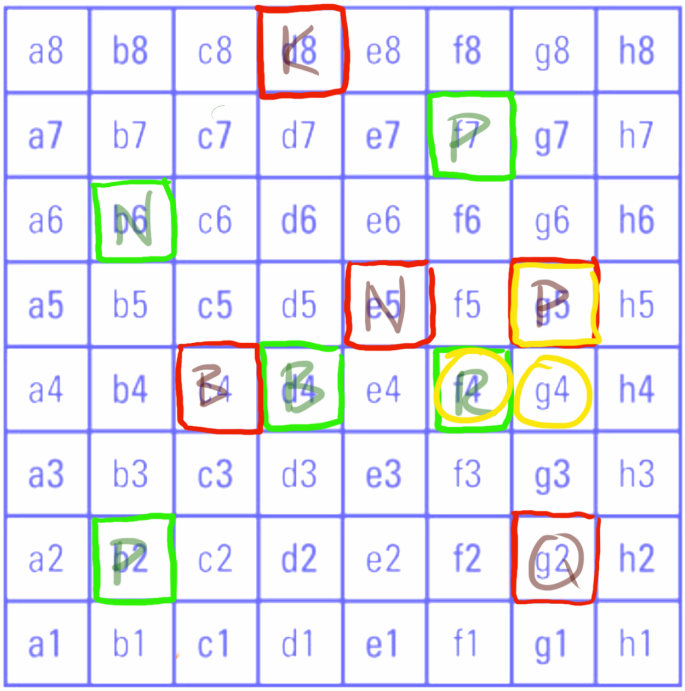
Program output was:

```
WHITE: Pf7, Nb6, Bd4, Rf4, Pb2
BLACK:  Kd8, Bc4, Ne5, Pg5, Qg2
PIECE TO MOVE: Ne5
LEGAL MOVES FOR BLACK Ne5: [g6, f7, c6, d7, g4, f3, d3]
```

Result: program output matched the expected output of Test 8.

3.2.9 Test 9

The next PIECE TO MOVE is Pg5, a black pawn. Black pawns may only move downward or diagonally capture a white piece downward. The image below shows the allowable moves to be [g4, f4].



Input for this test case was:

WHITE: Pf7, Nb6, Bd4, Rf4, Pb2
BLACK: Kd8, Bc4, Ne5, Pg5, Qg2
PIECE TO MOVE: Pg5

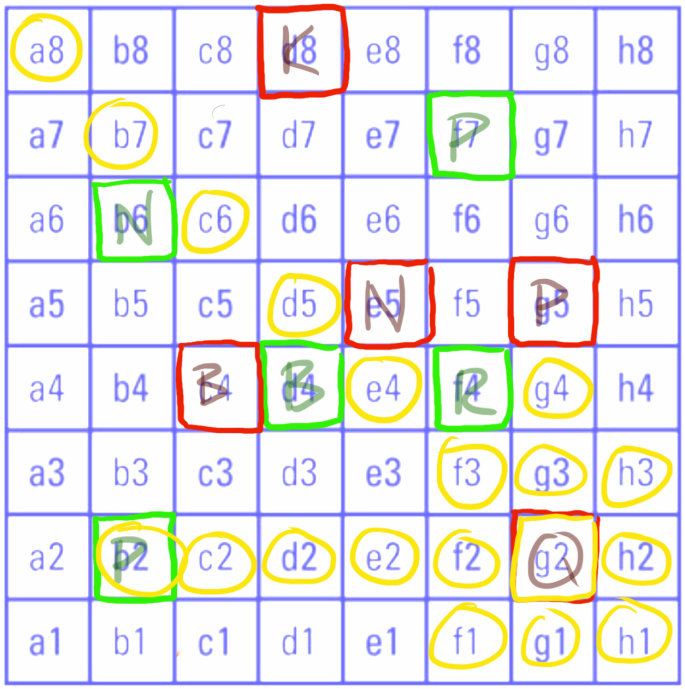
Program output was:

```
WHITE: Pf7, Nb6, Bd4, Rf4, Pb2
BLACK: Kd8, Bc4, Ne5, Pg5, Qg2
PIECE TO MOVE: Pg5
LEGAL MOVES FOR BLACK Pg5: [g4, f4]
```

Result: program output matched the expected output of Test 9.

3.2.10 Test 10

The last PIECE TO MOVE is Qg2, a black queen. Queens can move forward, backward, right, left, and diagonally until they reach the edge of the board or another piece. The allowable moves for a black queen at Qg2 are [f3, e4, d5, c6, b7, a8, g3, g4, g1, h2, f1, h3, h1, f2, e2, d2, c2, b2].



Input for this test case was:

WHITE: Pf7, Nb6, Bd4, Rf4, Pb2
BLACK: Kd8, Bc4, Ne5, Pg5, Qg2
PIECE TO MOVE: Qg2

Program output was:

```
WHITE: Pf7, Nb6, Bd4, Rf4, Pb2
BLACK: Kd8, Bc4, Ne5, Pg5, Qg2
PIECE TO MOVE: Qg2
LEGAL MOVES FOR BLACK Qg2: [g3, g4, g1, f2, e2, d2, c2, b2, h2, h3, h1, f3, e4, d5, c6, b7, a8, f1]
```

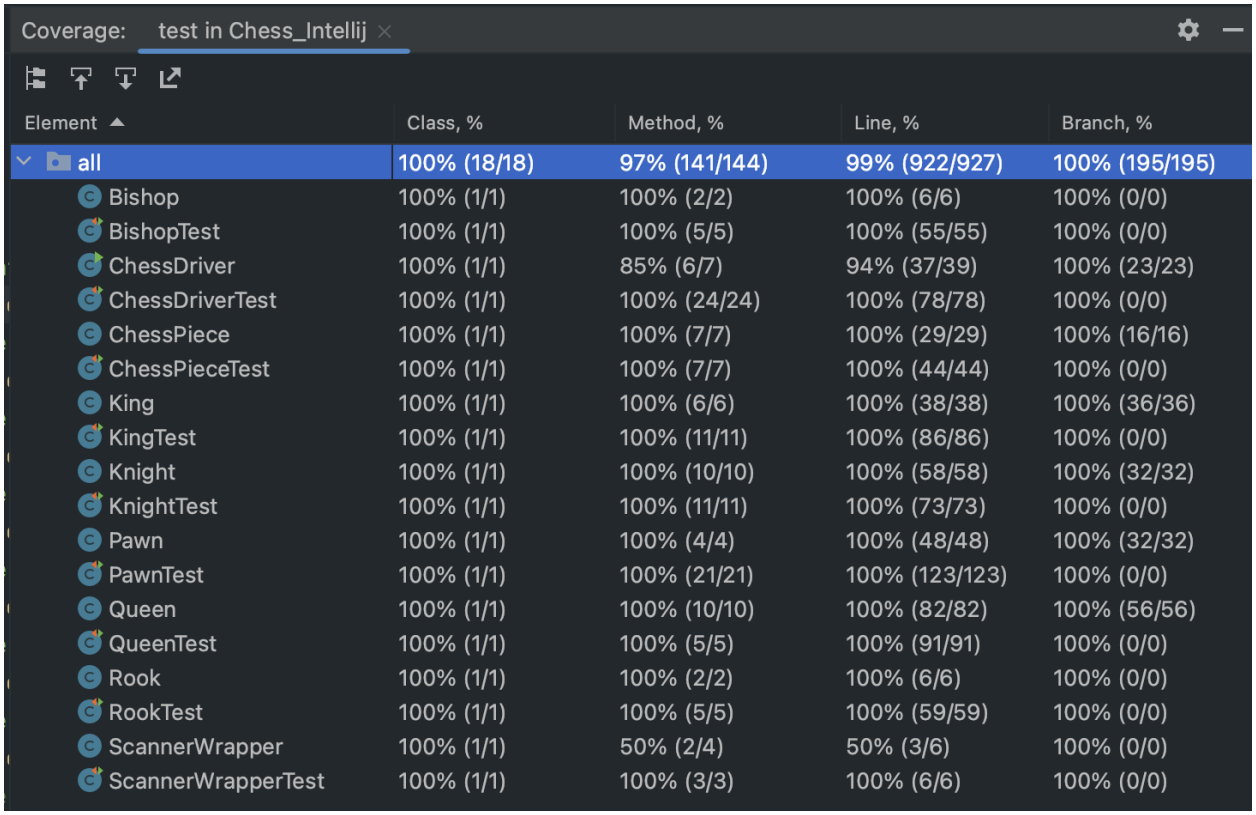
Result: program output matches the expected output for Test 10

4 Code Coverage

Including unit tests and system tests, a total of 69 tests were developed for the Chess Validation Engine. To test code coverage, Intellij's built-in code coverage tool was used. Before running the code coverage analysis, the Board class was removed from the system because it is never used in the functionality of the program (it was originally created to have a visual representation of a chess board while creating the functionality of the classes, and was no longer needed in the final version of the system). As mentioned, static analysis was run again after removing the Board class.

Code coverage analysis was run initially to gauge the sufficiency of the existing tests (at that point, there were 10 system tests and 38 unit tests), and yielded branch coverage of 80%. However, this coverage statistic covered all of the test classes in addition to the classes in the software system, so the value was skewed. Additional tests were written to cover the remaining uncovered branches in the system.

The results of the final code coverage report are shown below:



Element	Class, %	Method, %	Line, %	Branch, %
all	100% (18/18)	97% (141/144)	99% (922/927)	100% (195/195)
Bishop	100% (1/1)	100% (2/2)	100% (6/6)	100% (0/0)
BishopTest	100% (1/1)	100% (5/5)	100% (55/55)	100% (0/0)
ChessDriver	100% (1/1)	85% (6/7)	94% (37/39)	100% (23/23)
ChessDriverTest	100% (1/1)	100% (24/24)	100% (78/78)	100% (0/0)
ChessPiece	100% (1/1)	100% (7/7)	100% (29/29)	100% (16/16)
ChessPieceTest	100% (1/1)	100% (7/7)	100% (44/44)	100% (0/0)
King	100% (1/1)	100% (6/6)	100% (38/38)	100% (36/36)
KingTest	100% (1/1)	100% (11/11)	100% (86/86)	100% (0/0)
Knight	100% (1/1)	100% (10/10)	100% (58/58)	100% (32/32)
KnightTest	100% (1/1)	100% (11/11)	100% (73/73)	100% (0/0)
Pawn	100% (1/1)	100% (4/4)	100% (48/48)	100% (32/32)
PawnTest	100% (1/1)	100% (21/21)	100% (123/123)	100% (0/0)
Queen	100% (1/1)	100% (10/10)	100% (82/82)	100% (56/56)
QueenTest	100% (1/1)	100% (5/5)	100% (91/91)	100% (0/0)
Rook	100% (1/1)	100% (2/2)	100% (6/6)	100% (0/0)
RookTest	100% (1/1)	100% (5/5)	100% (59/59)	100% (0/0)
ScannerWrapper	100% (1/1)	50% (2/4)	50% (3/6)	100% (0/0)
ScannerWrapperTest	100% (1/1)	100% (3/3)	100% (6/6)	100% (0/0)

Class and branch coverage is 100%, while method and line coverage are slightly lower. Because ScannerWrapper creates a singleton of the Scanner class, some of the methods and lines are untestable, but the functionality of the class was tested and behaves as expected in the context of the system. The main method in the driver class is responsible for the uncovered lines and method in that class. The program compiles and executes, so the main method is functional and works correctly.

5 References

"Class Assert." *JUnit API Documentation*, JUnit, 1 Jan. 2020, https://junit.org/junit4/javadoc/4.13/org/junit/Assert.html.

"Class Scanner - Oracle Java Documentation." *Scanner (Java Platform SE 7)*, Oracle, 24 June 2020, https://docs.oracle.com/javase/7/docs/api/java/util/Scanner.html.

Dotnet-Bot. "Encoding.Default Property." *Microsoft .NET*, Microsoft, https://learn.microsoft.com/en-us/dotnet/api/system.text.encoding.default?view=net-7.0.

"Junit Test for System.out.println()." *Stack Overflow*, 1 June 2009, https://stackoverflow.com/questions/1119385/junit-test-for-system-out-println.

"Mockito and JUnit 5." *Baeldung*, 23 Feb. 2023, https://www.baeldung.com/mockito-junit-5-extension.

"Rules of Chess." *Wikipedia*, Wikimedia Foundation, 14 Jan. 2023, https://en.wikipedia.org/wiki/Rules_of_chess.

"SpotBugs Manual - Bug Descriptions." *SpotBugs 4.7.3 Documentation*, University of Maryland, https://spotbugs.readthedocs.io/en/latest/bugDescriptions.html#.