**CS 3331 Advance Object-Oriented Programming**

**Spring 2020**

**Lab Report #1**

Due: January 27, 2020

Professor: Dr. Oscar Mondragon

TA: Alireza Nouri

**Introduction**

Lab 1: Validating a Move for a Chess Piece focuses on reading various chess pieces from a file, creating chess piece objects, storing the objects in a array, asking the user for a new position, and validating whether the objects can move to that position. This lab was crucial to demonstrate the importance of inheritance and polymorphism by preventing students from using these aspects. Instead, students were only able to create a simple class with the chess piece attributes. All other functions occurred outside the class and required passing objects as parameters.

**Proposed Solution Design and Implementation**

The overall flow of my program focuses on reading a file, creating objects, storing the objects in an array, retrieving the new piece position from the user, and validating if each chess piece can move to that position. Initially, I created a base class called ChessPiece and derived classes for each of the chess types. I then used method overriding to verify the positions. However, after it was learned that there could only be a simple class that represents the C equivalent of a structure, I had to remove all these classes and simplify my program to include a ChessPiece class only. At the start of my program, the user inputs the path for where the .txt file is located with the chess pieces. The file is then read line by line. Each non-whitespace line is then evaluated. A series of checks are then made to ensure that the chess type, color (either black or white), x position, and y position of the chess piece are included. If so a ChessPiece object is created and added to an array list. I decided to use an array list since the number of pieces is unknown. Thus rather than having to read the file twice to count the number of lines, I decided to use the array list which could expand as needed. However, the assignment requires that the pieces eventually are stored in an array. Thus, I copy each object from the array list to the array after the entire file has been read. This results in an overall runtime complexity of O(n). After my array is populated, I ask the user to enter the new x and y position. The array is then iterated through and the function validateMove is called. The validateMove function determines which validate[ChessPiece Type]Move function to call based on the object’s name. Then based on the rules of chess, the new position is evaluated. For example, the queen can move diagonally, horizontally, or vertically. Thus, a corresponding canMoveDiagonal, canMoveVertical, and canMoveHorizontal are called to validate the new move. The program will continue to iterate through the objects and print whether they can move to the new position from their current position.

**Experimental Results**

**File Error Handling**

A screenshot of a cell phone

Description automatically generated

The screen shot above demonstrates that if a file could not be found, the program will terminate and alert the user. Additionally, I also check the file path to ensure that it contains a .txt extension. This is to prevent the program from opening a .xls or .jpg file instead. My program alerts the user and terminates the program as shown above.

**Testing the Instruction File**

chess.txt

A screenshot of a social media post

Description automatically generated

The screenshot above tests the instruction file provided. The test demonstrates that given the new position of D, 5, only the Queen could move to it horizontally. The pawn is unable to since it can only move vertically. The rook, knight, bishop, and king would all have to move a combination of vertical and horizontal squares to reach the new position. Since this violates their movement rules, the program correctly states that they cannot move.

**Testing the Chess Piece File Format**

initialPosition.txt

A screenshot of a social media post

Description automatically generated

The screenshot above demonstrates that empty lines are ignored by the program. Additionally, the program is not case sensitive as seen by the “WHiTE” input. However, the program does check to ensure that the x and y positions are valid, which is why the chess pieces on lines 2 and 3 were not created as objects. As for the object on line 6, the color was blue. This was an important design decision that I made. When reading online, I discovered that chess pieces are referred to as white or black. Thus, colors like pink, red, or green, are not considered valid chess pieces. Lastly, the screenshot shows that if the new position matches the current position, the move will not be valid since the piece is already located in the new position.

**Testing the User’s New Position**

A screenshot of a cell phone

Description automatically generated

As the screenshot demonstrates above, the user must provide a valid new position. Providing an x position not in the range of A to H or a y position not in the range of 1 to 9 will cause the program to terminate. Furthermore, If the user inputs a number when the program is expecting a character for the x-position, the user is notified, and the program terminates. Additionally, if the user provides a non-integer y-position, the InputMismatchException is caught and the program terminates.

**Pawn Test:**

pawnTestNewPosition.txt

A screenshot of a social media post

Description automatically generated

The screenshot above shows the different types of positions that a pawn could encounter. Since pawns can only move forward, it was crucial to test all three types of moves (horizontally, vertically, and diagonally) that could occur. Since the position was B 3, the white pawns at B 2 and B 1 could move up one or two spaces. For the black pawns, those at B 4 and B5 could move down the board, which is forward from their perspective. Additionally, the pawns at C 3 all proved that the pawns could not move horizontally. The pawns at C 2 proved that the pieces could not move diagonally. Thus, the pawn’s verification acted as expected.

**Rook Test:**

rookTestNewPosition.txt

A screenshot of a social media post

Description automatically generated

The screenshot above shows the different types of positions that a rook could encounter. A rook can move any number of squares vertically or horizontally, thus the piece color does not matter. The rooks at C 7 and C1 both demonstrate the programs ability to verify that the new position at C 6 is plausible since the pieces only need to move vertically. The rooks at H, 6 and B, 6 are also correct since the rooks can move horizontally. As for the rooks at position A, 8 and D, 1, the position C, 6 is not valid. In order to get to these positions, the rooks would have to move diagonally or vertically and horizontally. Thus, the computer correctly printed their inability to move.

**Bishop Test:**

bishopTestNewPosition.txt

A screenshot of a social media post

Description automatically generated

The screenshot above shows the different types of positions that a bishop could encounter. A bishop can only move diagonally which is why the first four clusters of chess pieces can move to the position B, 7. These tests demonstrate that all four directions of diagonals can accurately be verified. As for the pieces that cannot move, the pieces at B, 1 would require a vertical move, which is invalid. The pieces at E, 7 would require a horizontal move, which is also invalid. As for the pieces in C, 5 a combination of horizontal and diagonal moves would be required. Thus, my program correctly verified all the position accordingly.

**Queen Test:**

queenTestNewPosition.txt

A screenshot of a social media post

Description automatically generated

The screenshot above shows the different types of positions that a queen could encounter. The first cluster demonstrates the queen’s ability to move vertically, the third cluster demonstrates the queen’s ability to move horizontally, and the third cluster represents the queen’s ability to move diagonally. As for the fourth cluster, the queen cannot move from C, 4 to B, 2 without do a combination or vertical and horizontal moves. Thus, the program correctly recorded the queen’s inability to move.

**King Test:**

kingTestNewPosition.txt

A screenshot of a social media post

Description automatically generated

The screenshot above shows the different types of positions that a king could encounter. The first cluster represents the king’s ability to move vertically by one square. However, the second cluster shows that the king cannot move more than one square vertically. The third cluster shows the king’s ability to move horizontally by one square. The fourth cluster demonstrates the king’s inability to move more than one square horizontally since the king would have to move three spaces. The fourth cluster demonstrates the king’s ability to move diagonally by one unit. However, for the pieces at A, 6, they would be unable to reach C, 4 since they would have to move 2 spaces diagonally. Thus, my program verified the positions as expected.

**Knight Test:**

knightTestNewPosition.txt

A screenshot of a social media post

Description automatically generated

The screenshot above shows the different types of positions that a knight could encounter. The first cluster represents all knight moves that require one square horizontally and two squares vertically to be taken. The second cluster represents all knight moves that require two squares horizontally and one square vertically to be taken. In all instances an L-shape is taken to reach the new position D, 6. The screenshot demonstrates that my tests accurately verified the positions. The third cluster demonstrates that a knight cannot move horizontally, and the fourth cluster demonstrates that a knight cannot move vertically. Lastly, my program was able correctly verify that a knight cannot move diagonally through the third cluster of pieces.

**Conclusion**

In all, I learned from the lab the importance of inheritance and polymorphism. Since I had originally created the lab using derived classes, I had to remove these classes and implement a single struct-like class. In doing so, I was able to clearly identify why programming languages like C++ were created. By having more functions that are not nested in classes, the program becomes more unorganized. Additionally, I found myself having to scroll through code for a particular function when I could have easily found the function in a class. Lastly, my program was able to fulfill the task of the lab and verify the new positions of the user provided chess pieces.

**Appendix**