San Francisco University of Quito

Final project

Advanced Programming in C++

Benjamin Dillon Gangotena 00325654

Nathalia Mercedes Martínez Torres 00320858

Move method logic

Regarding the logic of the move method we did the following:

1. Coordinate Translation:

```
    vector<int> coordenadas = TraducirInput(origen, destino);
    int origenX = coordenadas[0];
    int origenY = coordenadas[1];
    int destinoX = coordenadas[2];
    int destinoY = coordenadas[3];
```

The method begins by translating the origin and destination coordinates using a function called`TranslateInput`. This step assumes that there is some prior logic to convert the source and destination input to numerical coordinates.

2. Coordinate Validation:

```
// Verificar si las coordenadas están dentro del rango del tablero
if (origenX < 0 || origenX >= 8 || origenY < 0 || origenY >= 8 ||
    destinoX < 0 || destinoX >= 8 || destinoY < 0 || destinoY >= 8) {
    cout << "Coordenadas fuera de rango." << endl;
    return;
}
```

It is checked if the coordinates are within the range of the board (8x8). If any of the coordinates are outside this range, an error message is printed and the function terminates.

3. Obtaining the Piece in the Origin Position:

```
// Obtener la pieza en la posición de origen
Pieza* pieza = tablero[origenX][origenY];
```

The piece is obtained in the original position of the board.

4. Verification of Part Existence in the Box of Origin:

```
// Verificar si hay una pieza en la casilla de origen
if (pieza == nullptr) {
    cout << "No hay pieza en la casilla de origen." << endl;
    return;
}
```

It is checked if there is a piece in the origin box. If the box is empty (i.e. nullptr), an error message is printed and the function terminates.

5. Motion Validation for the Piece:

```
// Verificar si el movimiento es válido para la pieza
if (pieza->MovimientoValido(origenX, origenY, destinoX, destinoY)) {
```

It is verified if the movement is valid for the piece. This logic depends on the specific implementation of the classPartand his methodValidMovement.

6. Successful Move or Error:

```
    // Verificar si la casilla de destino está vacía
    if (tablero[destinoX][destinoY] == nullptr) {
        // Realizar el movimiento si la casilla de destino está vacía
        tablero[destinoX][destinoY] = pieza;
        tablero[origenX][origenY] = nullptr;
        cout << "Movimiento exitoso." << endl;
    } else {
        // La casilla de destino está ocupada, no se puede realizar el movimiento
        cout << "Movimiento no válido. La casilla de destino está ocupada." << endl;
    }
} else {
    // Movimiento no válido para la pieza
    cout << "Movimiento no válido para la pieza." << endl;
}
```

If the target space is empty, the piece is moved to that position and the board is updated. A success message is printed.

If the destination space is occupied, an error message is printed as the move cannot be made.

If the move is not valid for the part, an error message is printed.

In summary, the methodMoveIt verifies the coordinates, the existence of a piece in the origin square, the validity of the movement for that piece and finally makes the movement or not according to the occupation of the destination square.

Movement Logic Valid in each class

ValidMovement method in the Pawn class:

This method checks if the move is valid for a pawn. For white pawns, it allows moving one square forward (destinationY == originY + 1) or two squares on the first move (originY == 1 && destinationY == originY + 2). It does not allow movements in other directions, and it does not allow it to make jumps like the horse. It is not possible for the pawn to eat other pieces on the board.

ValidMovement method in the Tower class:

Check if the move is valid for a tower. For white towers it only allows movements in a straight line, either horizontal (originX == destinationX && originY != destinationY) or vertical (originY == destinationY && originX != destinationX). It does not allow other movements, nor does it allow you to jump like a horse. And analogously to the pawn, it is not allowed to eat other pieces on the board.

ValidMovement method in the Horse class:

This method checks if the move is valid for a horse. Only if the horse is white, implement the logic for "L" movements (deltaX == 2 && deltaY == 1 || deltaX == 1 && deltaY == 2), that is, obviously the horse token can jump over other tokens. What this does not allow is that it goes in a straight line, diagonal, etc. Only allows jumps that are

characteristic of the movement of the knight in the game of chess, in L in all directions. It does not allow the horse to eat other pieces.

ValidMovement method in the Bishop class:

This method checks if the move is valid for a bishop. If the bishop to move is a bank, this method allows diagonal moves (deltaX == deltaY). It does not allow this piece to eat other pieces, and it also does not allow moves that are not characteristic of a bishop in chess, such as straight lines or jumping.

ValidMovement method in the Lady class:

This method combines the logic of rook and bishop moves. It allows horizontal, vertical and diagonal movements, but does not allow movements like the jumping knight on other pieces. The queen is literally the queen of the board and with this function she can go forward, backward, left, right, diagonally and can even go backwards. Due to project instructions, the monarch is also not allowed to eat other pieces on the board in this method.

ValidMovement method in the King class:

This method in the King class allows, if and only if the piece is white, movements in any direction advancing one square (deltaX <= 1 && deltaY <= 1). That is, the king can go in all directions, but only one square, as expected from his characteristic move on the chess board. However, this one cannot eat other pieces, despite being a King. He also can't jump over other pieces.

Challenges faced:

1. **Polymorphism and Class Design:**

   Implementing a chess game in code involved dealing with multiple types of pieces, each with their own rules of movement. The effective use of polymorphism and class hierarchy was crucial to the development of the game. Another challenge was ensuring

that the derived classes (Pawn, Rook, Knight, etc.) adhere correctly to the interface defined by the base class (Piece).

2. Validation of Movements:

It was difficult to ensure that each piece followed the correct chess rules when validating moves. Well, for each piece, the specific logic for each type of piece had to be implemented in each corresponding class and each piece had to be made to move only when it was white. Telling the program that only the white pieces should be moved was even more complicated than coding so that all the pieces could be moved.

Learned lessons:

1. Abstraction and Modular Design:

Through the project we learned how to design classes and methods that are easily understandable and modular. Since in the project each class had to clearly represent a piece in the game, and the methods must perform specific tasks for each of the pieces.

2. Testing and Debugging:

Throughout the project we learned the importance of extensive testing to ensure that each piece moves according to the rules of chess and that the board updates correctly. Since we had to try different ways to make sure that the pieces that should be moved, only the white ones, moved correctly and the black ones did not move at all. So debugging becomes crucial to find and fix errors.

3. Error and Exception Handling:

To maintain control of errors it was necessary to implement error and exception handling to ensure that the program does not fail in unexpected situations. So clear error messages were provided to facilitate troubleshooting.

4. User Interface:

Through the user console in the main part of the file, we appreciate that user interaction is key in a game. The design of a user interface must be clear and easy to understand, as well as the proper management of user input, these aspects are important and should be considered especially when creating a game or an application that is in constant contact. with the user.

5. Optimization and Efficiency:

We observe that as the program becomes more complex, code optimization and efficiency become more relevant. So what remains is for us to look for opportunities to improve performance, especially in larger games or with more features.

6. Design Flexibility:

The work sought to keep the code design flexible enough to allow future expansions or modifications without rewriting large parts of the code. Since in this way we learned that correcting errors, as well as adding features or improvements, is much easier.