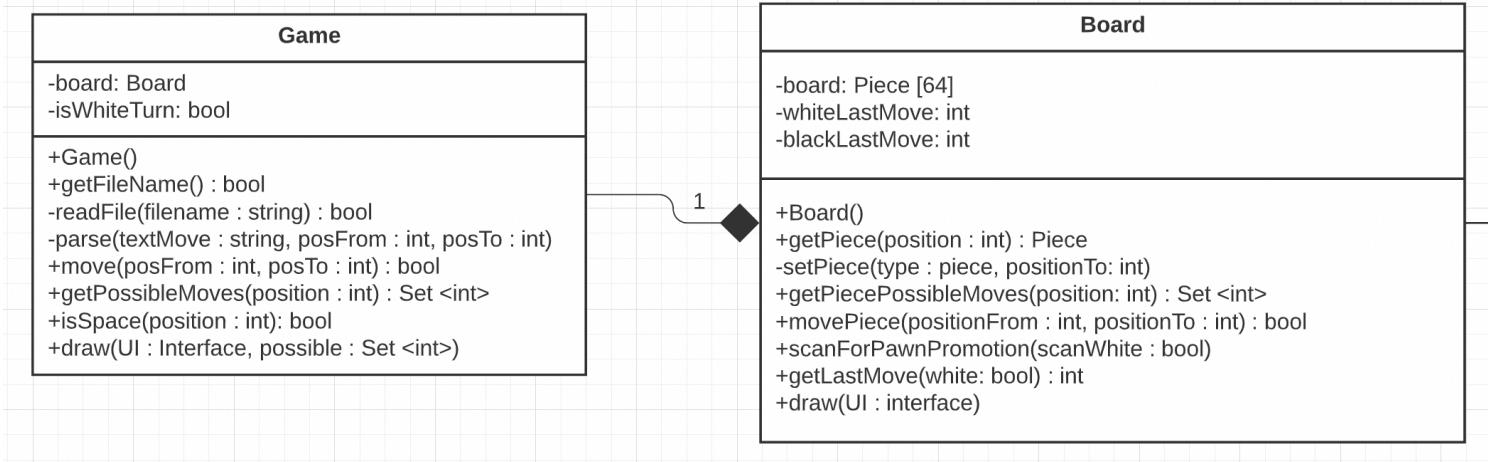


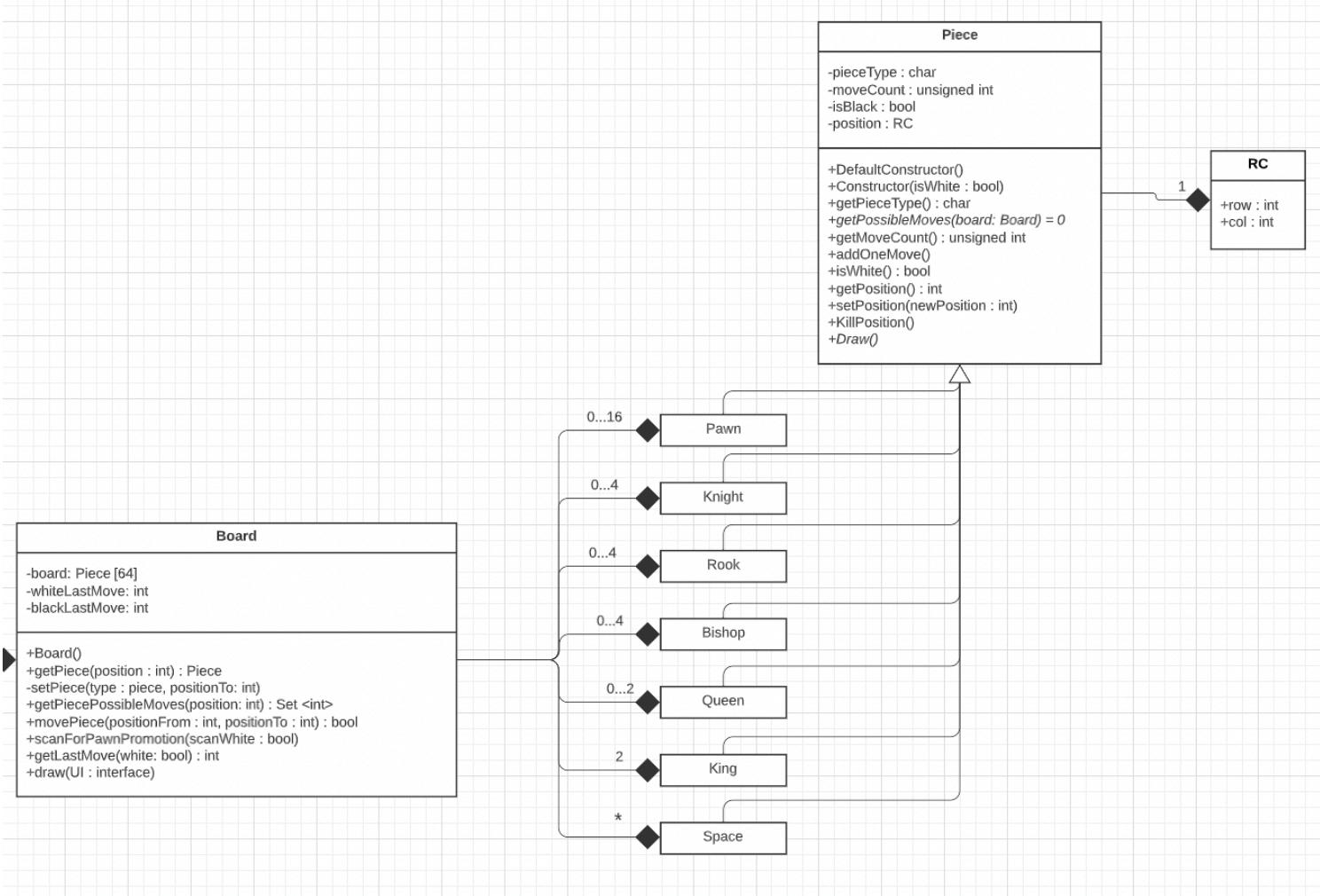
LAB 02: Chess Design

Class Diagrams:

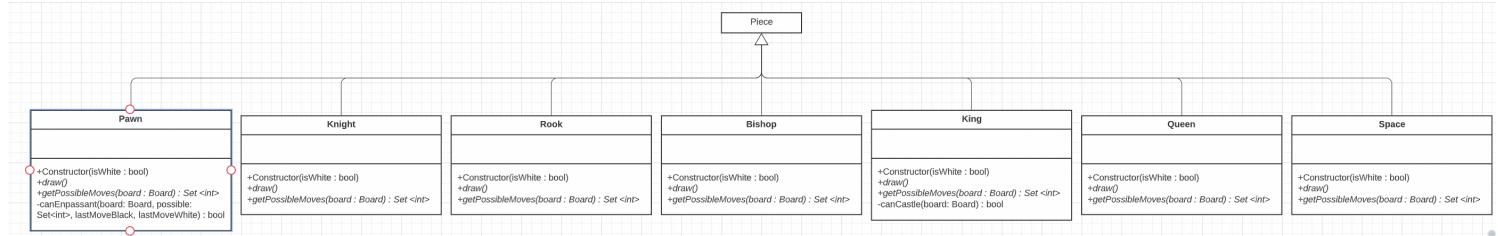
The Game and Board class relation.



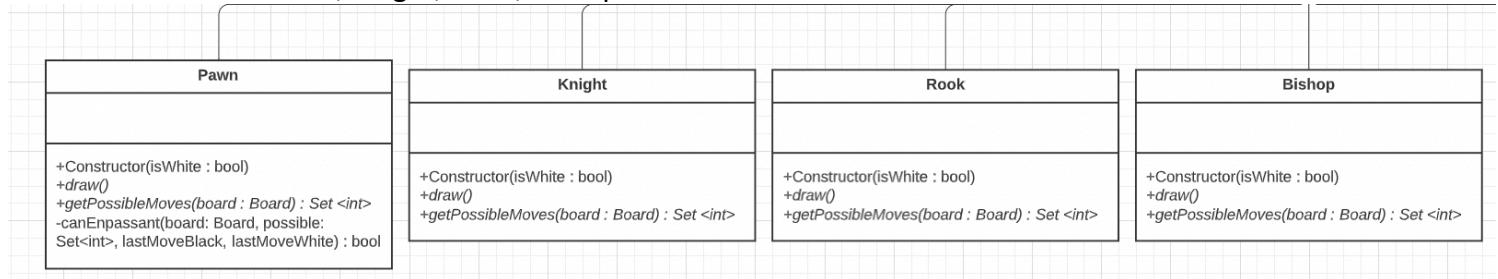
The Board and Piece class relation.



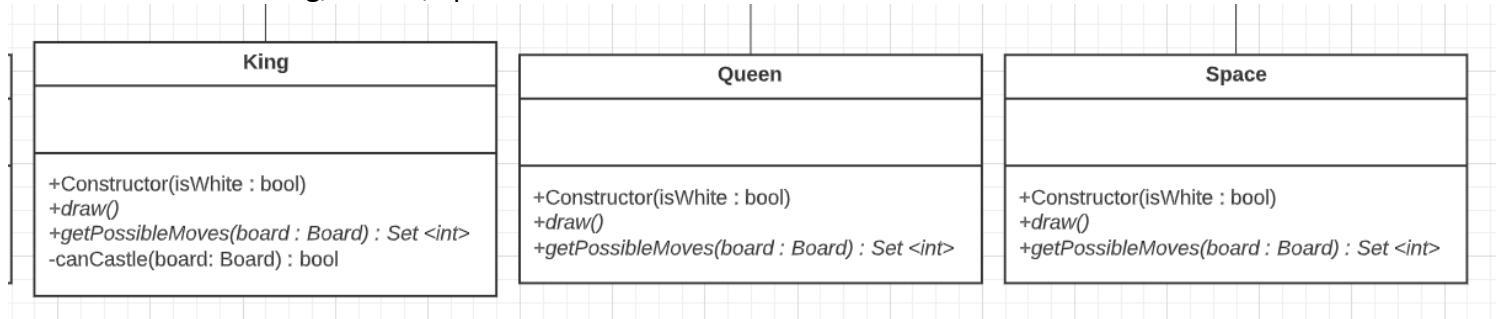
The Derived Piece Classes: Pawn, Knight, Rook, Bishop, King, Queen, Space.



A closer look at the Pawn, Knight, Rook, Bishop classes.

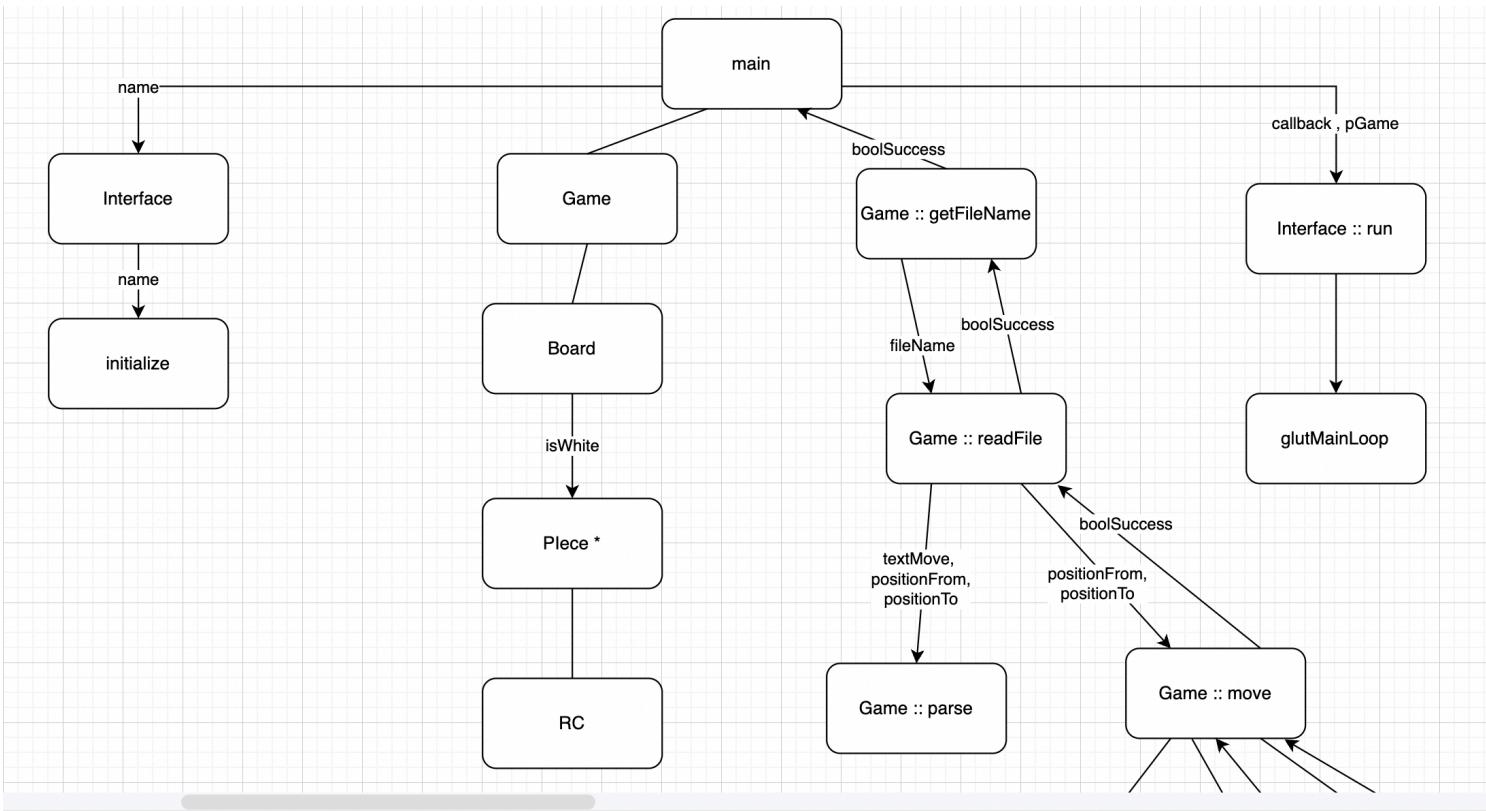


A closer look at the King, Queen, Space classes.

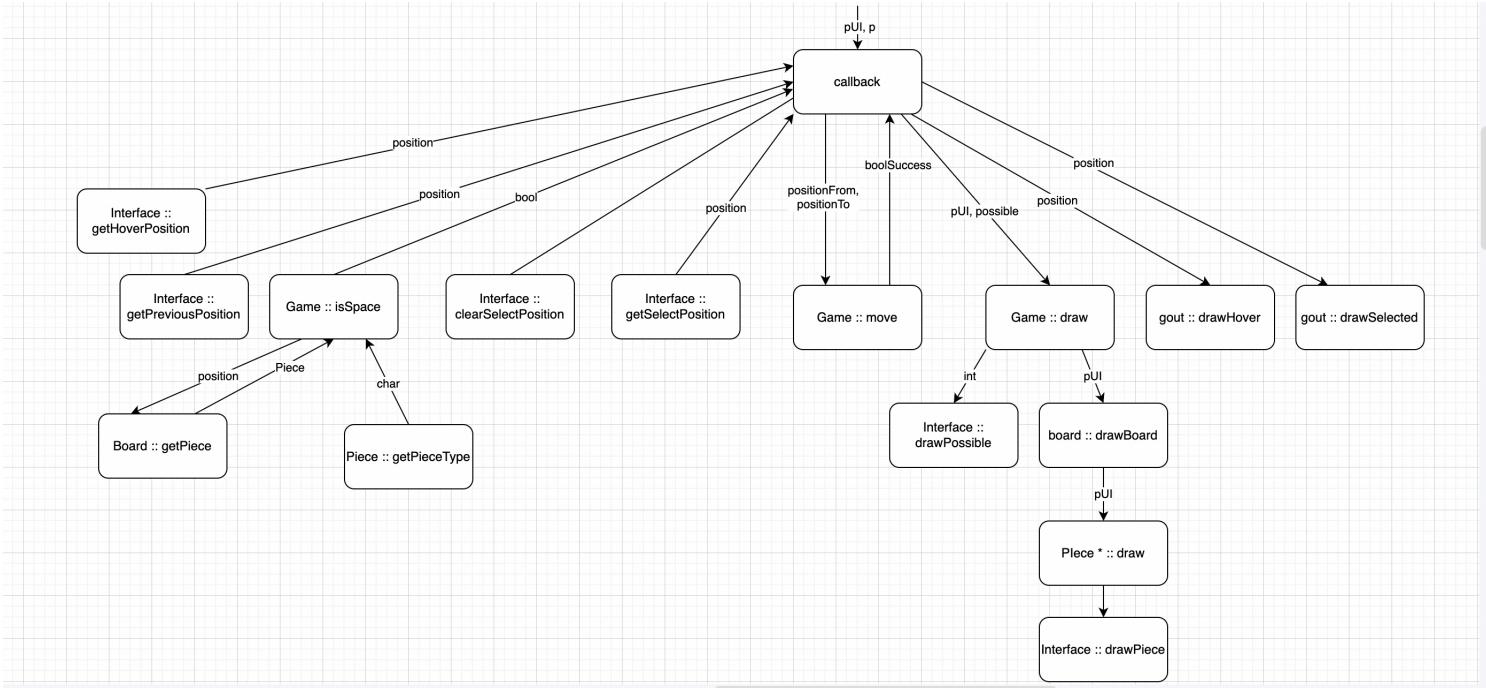


Structure Chart:

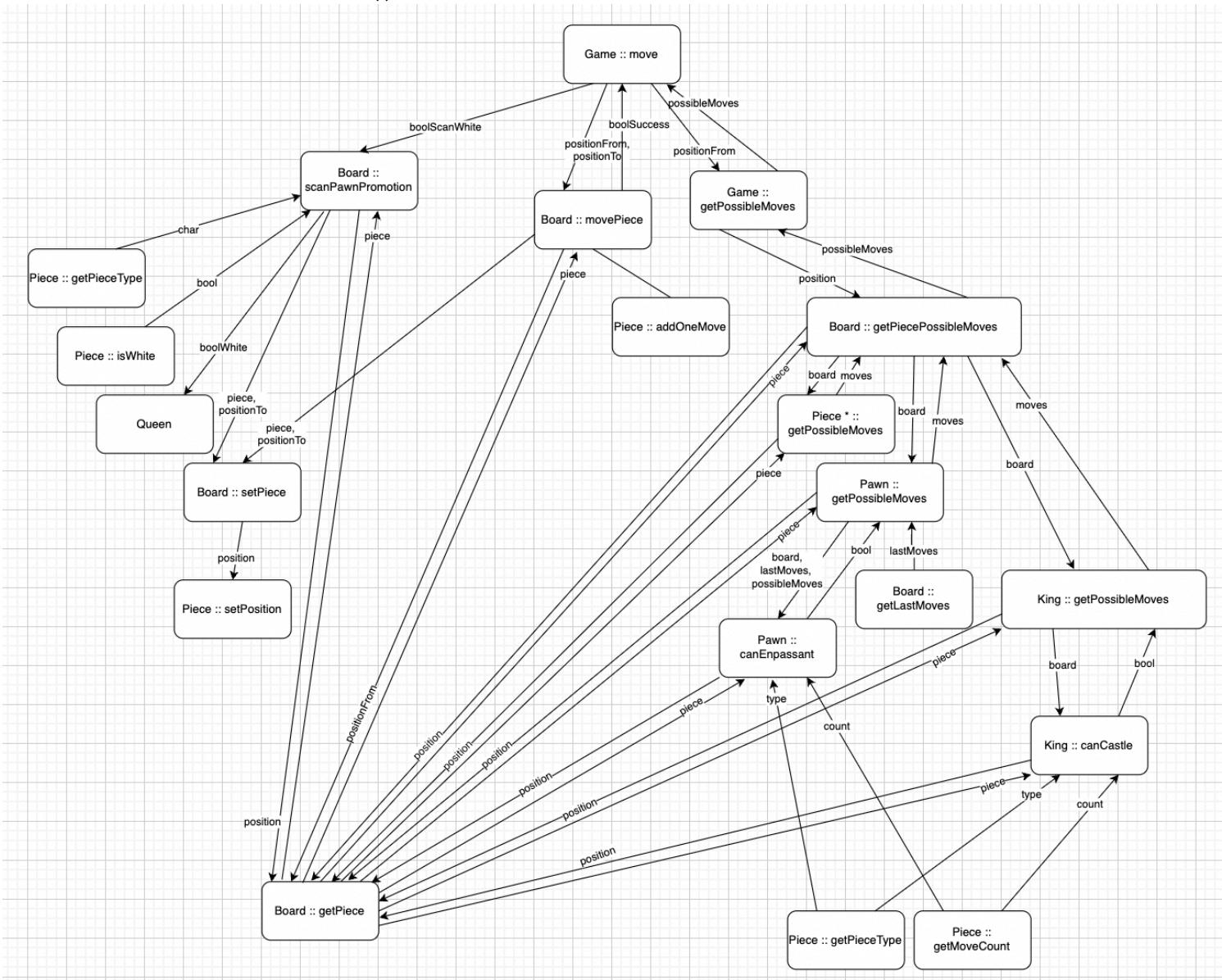
Everything that happens before OpenGL is called.



`glutMainLoop()` runs the OpenGL graphics loop; we do not get to see how it works. It does periodically call `callback()`.



A closer look at Game :: move()



Pseudocode:

Castling:

```
1 King :: canCastle(board)
2
3     // Set the correct positions to check based on King color
4     IF isBlack
5         // Black King
6         correctKingPosition <- 60
7         correctRookPosition <- 63
8         rookShouldBeWhite <- FALSE
9     ELSE
10        // White King
11        correctKingPosition <- 4
12        correctRookPosition <- 7
13        rookShouldBeWhite <- TRUE
14
15    // First check if the King is in the correct position and has not moved
16    IF getPosition() = correctKingPosition AND moveCount = 0
17
18        // Next check that the next two positions are spaces
19        nextSquare <- board.getPiece(correctKingPosition + 1)
20        twoSquaresOver <- board.getPiece(correctKingPosition + 2)
21        IF nextSquare.getPieceType() = 's' AND twoSquaresOver.getPieceType() = 's'
22
23            // Lastly check that the rook is in the correct position and has not moved
24            piece <- board.getPiece(correctRookPosition)
25            IF piece.getPieceType() = 'r' AND piece.getMoveCount() = 0 AND piece.isWhite() = rookShouldBeWhite
26
27                // return true to the client; the king can castle.
28                // In this case the only client is King :: getPossibleMoves()
29                // and having received the return value true, getPossibleMoves()
30                // will add the castle move to the set of possible moves for the King.
31                RETURN TRUE
32
33            ELSE
34                RETURN FALSE
35        ELSE
36            RETURN FALSE
37    ELSE
38        RETURN FALSE
39
```

En-Passant:

```
1  Pawn :: canEnPassant(*possibleMoves, lastWhiteMove, lastBlackMove ,Board)
2      // Set values for the current pawn
3      SET startPos <- piece.getPosition()
4
5      // Checking if piece is on the sides
6      switch(startPos)
7          case: 24
8              SET left = FALSE
9              SET RIGHT = TRUE
10         case: 31
11             SET left = TRUE
12             SET RIGHT = FALSE
13         case: 40
14             SET left = FALSE
15             SET RIGHT = TRUE
16         case: 47
17             SET left = TRUE
18             SET RIGHT = FALSE
19     Default:
20         SET left = TRUE
21         SET RIGHT = TRUE
22
23     // Checking for black side
24     IF isBlack
25         IF left
26             board.getPieceType(position + 1) = 'p' && piece.getPiece(position + 1).getMoveCount() = 1;
27             IF (startPos + 1) == lastWhiteMove
28                 RETURN lastWhiteMove - 8
29         IF RIGHT
30             piece.getPieceType(position + 1) = 'p' && piece.getMoveCount() = 1;
31             IF (startPos + 1) == lastWhiteMove
32                 RETURN lastWhiteMove - 8
33     // Checking for White side
34     ELSE
35         IF left
36             board.getPieceType(position - 1) = 'P' && board.getPiece(position - 1).getMoveCount() = 1;
37             IF (startPos + 1) == lastBlackMove
38                 RETURN lastBlackMove + 8
39         IF RIGHT
40             board.getPieceType(position - 1) = 'P' && board.getPiece(position - 1).getMoveCount() = 1;
41             IF (startPos + 1) == lastBlackMove
42                 RETURN lastBlackMove + 8
43
44     RETURN FALSE
```

Pawn Promotion:

```
1 Board :: scanForPawnPromotion(scanWhite)
2     IF scanWhite
3         start <- 56
4         stop <- 63
5     ELSE
6         start <- 0
7         stop <- 7
8
9     FOR position <- start...stop
10        IF board.getPiece(position) = 'p' and board.getPiece(position).isWhite() = scanWhite
11            q <- Queen(scanWhite)
12            board.setPiece(Q, position)
13        RETURN
14
```

Test Cases:

```
1 #include <iostream>
2 #include <cassert>
3
4 using namespace std;
5
6 void testBoardMovePiece()
7 {
8     // Setup
9     Board board;
10    Pawn pawn(true);
11    // Exercise
12    board.movePiece(pawn, 16);
13    // Verify
14    assert(pawn.position == 16);
15    assert(board[16].getPieceType() == 'p');
16 } // Teardown
17
18 void testBoardScanPawnPromotion()
19 {
20     // Setup
21     Pawn pawn(true);
22     pawn.position = 58;
23     Board board;
24     // Exercise
25     board.scanPawnPromotion(true);
26     // Verify
27     assert(board[58].getPieceType() == 'q');
28 } // Teardown
29
30 void testBoardScanPawnNoPromotion()
31 {
32     // Setup
33     Pawn pawn(true);
34     pawn.position = 32;
35     // Exercise
36     board.scanPawnPromotion(true);
37     // Verify
38     assert(board[32].getPieceType() == 'p');
39 } // Teardown
40
```

```
40+
41 void testBoardGetPieceType()
42 {
43     // Setup
44     Knight knight(true);
45     knight.position = 20;
46     // Exercise
47     // Verify
48     assert(getPieceType(20) == 'n');
49 } // Teardown
50
51 void testBoardGetPossibleMoves()
52 {
53     // Setup
54     Pawn pawn(true);
55     pawn.position = 8;
56     // Exercise
57     // Verify
58     assert(pawn.getPossibleMoves() == {16});
59 } // Teardown
60
61 void testBoardDefault()
62 {
63     // Setup
64     Board board;
65     // Exercise
66     // Verify
67     assert(board == {
68         'r', 'n', 'b', 'q', 'k', 'b', 'n', 'r',
69         'p', 'p', 'p', 'p', 'p', 'p', 'p', 'p',
70         ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ',
71         ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ',
72         ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ',
73         ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ',
74         // ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ',
75         'P', 'P', 'P', 'P', 'P', 'P', 'P', 'P',
76         'R', 'N', 'B', 'Q', 'K', 'B', 'N', 'R'
77     });
78 } // Teardown
```

```

1 #include <iostream>
2 #include <cassert>
3
4 using namespace std;
5
6
7 void kingTestPossibleMoves()
8 {
9     // Setup
10    King king(true);
11    king.position = 35;
12    // Exercise
13    // Verify
14    assert(king.getPossibleMoves() ==
15    {26, 27, 28, 34, 36, 42, 43, 44});
16 } // Teardown
17
18 void kingTestDefaultWhite()
19 +
20 {
21     // Setup
22     King king(true);
23     // Exercise
24     // Verify
25     assert(king.position == 4);
26 } // Teardown
27
28 void kingTestDefaultBlack()
29 {
30     // Setup
31     King king(true);
32     // Exercise
33     // Verify
34     assert(king.position == 60);
35 } // Teardown

```

```

75
76 + void kingTestGetMoveCount()
77 {
78     // Setup
79     King king(true);
80     // Exercise
81     king.addOneForward();
82     // Verify
83     assert(king.moves == 1);
84 } // Teardown
85
86 void kingTestKillPositionForward() 99
87 {
88     // Setup
89     King king(true);
90     king.position = 20;
91     Pawn pawn(false);
92     pawn.position = 28;
93     // Exercise
94     king.killPosition(28);
95     // Verify
96     assert(king.position == 28);
97 } // Teardown

```

```

36 void kingTestCanCastle()
37 +
38 {
39     // Setup
40     King king(true);
41     Rook rook(true);
42     // Exercise
43     // Verify
44     assert(king.moves == 0);
45     assert(rook.moves == 0);
46     assert(king.position == 60);
47     assert(rook.position == 63);
48     assert(twoSquaresOver.getPieceType() == "s");
49     assert(nextSquare.getPieceType() == "s");
50 } // Teardown
51
52 void kingTestCantCastle()
53 {
54     // Setup
55     King king(true);
56     Rook rook(true);
57     // Exercise
58     // Verify
59     assert(king.moves == 0);
60     assert(rook.moves == 0);
61     assert(king.position == 60);
62     assert(rook.position == 63);
63     assert(twoSquaresOver.getPieceType() != "s");
64     assert(nextSquare.getPieceType() != "s");
65 } // Teardown
66
67 void kingTestAddOneForward()
68 {
69     // Setup
70     King king(true);
71     // Exercise
72     king.addOneForward();
73     // Verify
74     assert(king.position == 12);
75 } // Teardown

```

```

98
99 void kingTestKillDiagonalLeft()
100 {
101     // Setup
102     King king(true);
103     pawn.position = 20;
104     Pawn pawn(false);
105     pawn.position = 27;
106     // Exercise
107     king.killPosition(27);
108     // Verify
109     assert(king.position == 27);
110 } // Teardown

```