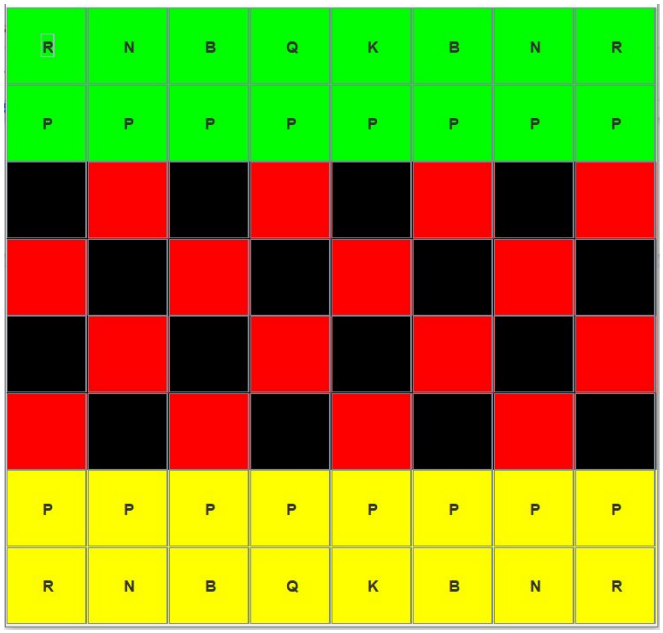
CSDS 132 Final Project Testing Report

@ author: Lam Nguyen **@ cwruID: ltn18**

Contents:

1) TEST FILLING BOARD

Test 1.1: initiate full chessBoard north and south (PASSED)

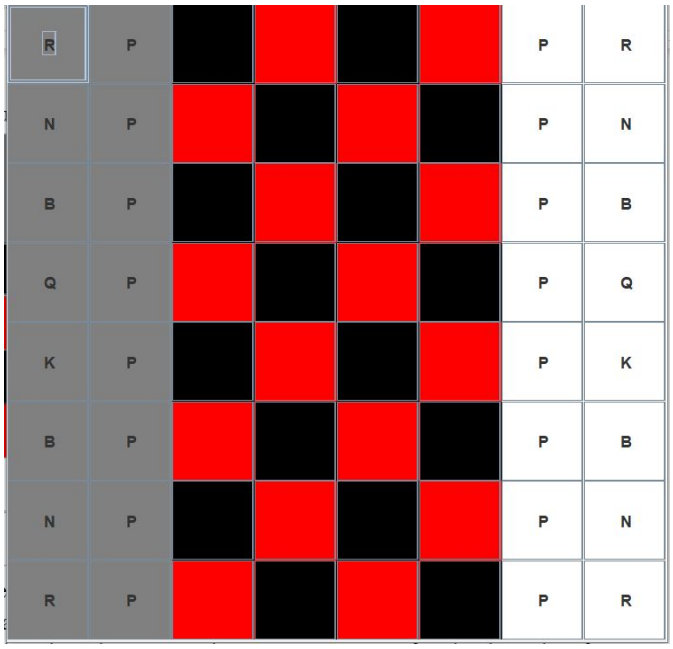


Description: We will add the pieces onto the board one by one. After doing so, we will get a table that has yellow representing south and green representing north.

JUnit: first = SOUTH & second = NORTH

```
// TODO: add PawnPiece
for (int i = 0; i < 8; ++i) {
    chessBoard.addPiece(new PawnPiece(second, chessBoard),  row: 1, i);
    chessBoard.addPiece(new PawnPiece(first, chessBoard),  row: 6, i);
}
// TODO: add KingPiece
chessBoard.addPiece(new KingPiece(second, chessBoard),  row: 0,  col: 4);
chessBoard.addPiece(new KingPiece(first, chessBoard),  row: 7,  col: 4);
// TODO: add QueenPiece
chessBoard.addPiece(new QueenPiece(second, chessBoard),  row: 0,  col: 3);
chessBoard.addPiece(new QueenPiece(first, chessBoard),  row: 7,  col: 3);
// TODO: add BishopPiece
chessBoard.addPiece(new BishopPiece(second, chessBoard),  row: 0,  col: 2);
chessBoard.addPiece(new BishopPiece(second, chessBoard),  row: 0,  col: 5);
chessBoard.addPiece(new BishopPiece(first, chessBoard),  row: 7,  col: 2);
chessBoard.addPiece(new BishopPiece(first, chessBoard),  row: 7,  col: 5);
// TODO: add KnightPiece
chessBoard.addPiece(new KnightPiece(second, chessBoard),  row: 0,  col: 1);
chessBoard.addPiece(new KnightPiece(second, chessBoard),  row: 0,  col: 6);
chessBoard.addPiece(new KnightPiece(first, chessBoard),  row: 7,  col: 1);
chessBoard.addPiece(new KnightPiece(first, chessBoard),  row: 7,  col: 6);
// TODO: add RookPiece
chessBoard.addPiece(new RookPiece(second, chessBoard),  row: 0,  col: 0);
chessBoard.addPiece(new RookPiece(second, chessBoard),  row: 0,  col: 7);
chessBoard.addPiece(new RookPiece(first, chessBoard),  row: 7,  col: 0);
chessBoard.addPiece(new RookPiece(first, chessBoard),  row: 7,  col: 7);
```

Test 1.2: initiate full chessBoard east and west (PASSED)



Description: We will add the pieces onto the board one by one. After doing so, we will get a table that has white representing east and gray representing west.

JUnit: first = EAST & second = WEST

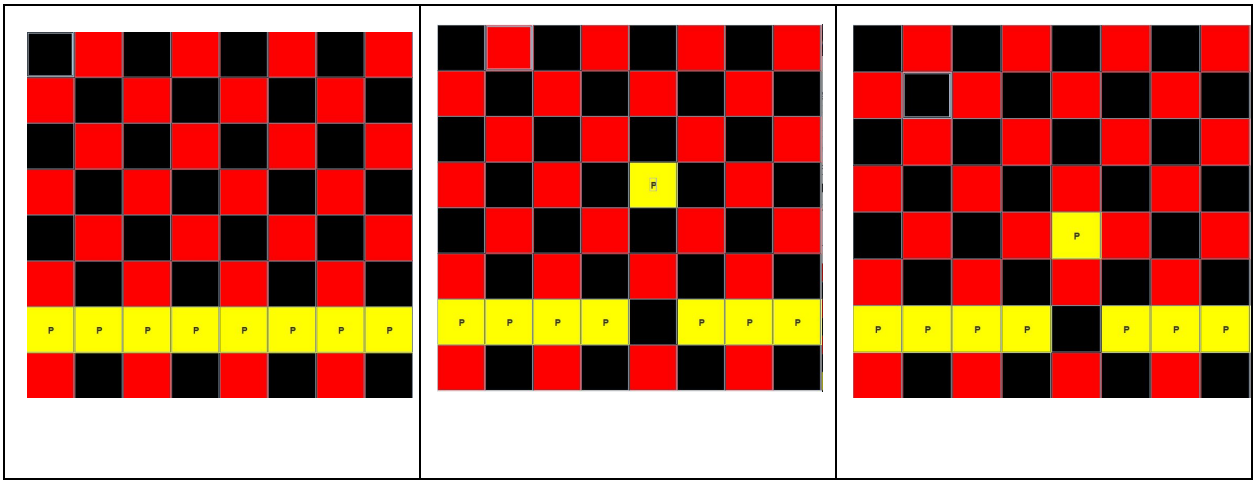
```
// TODO: add PawnPiece
for (int i = 0; i < 8; ++i) {
    chessBoard.addPiece(new PawnPiece(second, chessBoard), i,  col: 1);
    chessBoard.addPiece(new PawnPiece(first, chessBoard), i,  col: 6);
}
// TODO: add KingPiece
chessBoard.addPiece(new KingPiece(second, chessBoard),  row: 4,  col: 0);
chessBoard.addPiece(new KingPiece(first, chessBoard),  row: 4,  col: 7);
// TODO: add QueenPiece
chessBoard.addPiece(new QueenPiece(second, chessBoard),  row: 3,  col: 0);
chessBoard.addPiece(new QueenPiece(first, chessBoard),  row: 3,  col: 7);
// TODO: add BishopPiece
chessBoard.addPiece(new BishopPiece(second, chessBoard),  row: 2,  col: 0);
chessBoard.addPiece(new BishopPiece(second, chessBoard),  row: 5,  col: 0);
chessBoard.addPiece(new BishopPiece(first, chessBoard),  row: 2,  col: 7);
chessBoard.addPiece(new BishopPiece(first, chessBoard),  row: 5,  col: 7);
// TODO: add KnightPiece
chessBoard.addPiece(new KnightPiece(second, chessBoard),  row: 1,  col: 0);
chessBoard.addPiece(new KnightPiece(second, chessBoard),  row: 6,  col: 0);
chessBoard.addPiece(new KnightPiece(first, chessBoard),  row: 1,  col: 7);
chessBoard.addPiece(new KnightPiece(first, chessBoard),  row: 6,  col: 7);
// TODO: add RookPiece
chessBoard.addPiece(new RookPiece(second, chessBoard),  row: 0,  col: 0);
chessBoard.addPiece(new RookPiece(second, chessBoard),  row: 7,  col: 0);
chessBoard.addPiece(new RookPiece(first, chessBoard),  row: 0,  col: 7);
chessBoard.addPiece(new RookPiece(first, chessBoard),  row: 7,  col: 7);
```

EUROPEAN CHESS: We can see that the table for north and south vs east and west are transpose of each other, which makes rotating the board be switching the position of row and column. Therefore, we need only to consider one kind of them later in the test report. The following tests will contain specific pieces at designated positions for easy view. Also, we set the legalPieceToPlay to be true all the time.

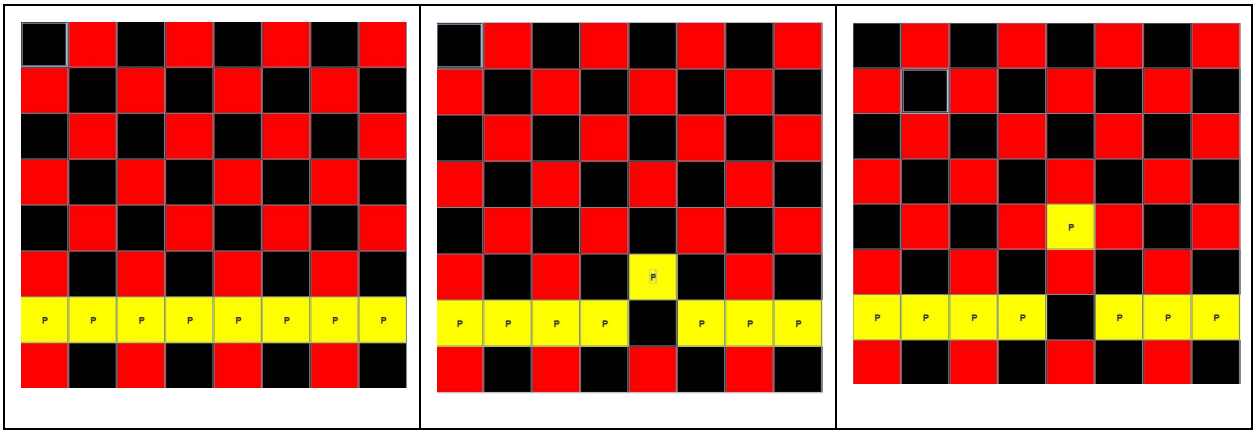
2) TEST EUROPEAN CHESS PIECE MOVEMENT

Test 2.1: test PawnPiece move (PASSED)

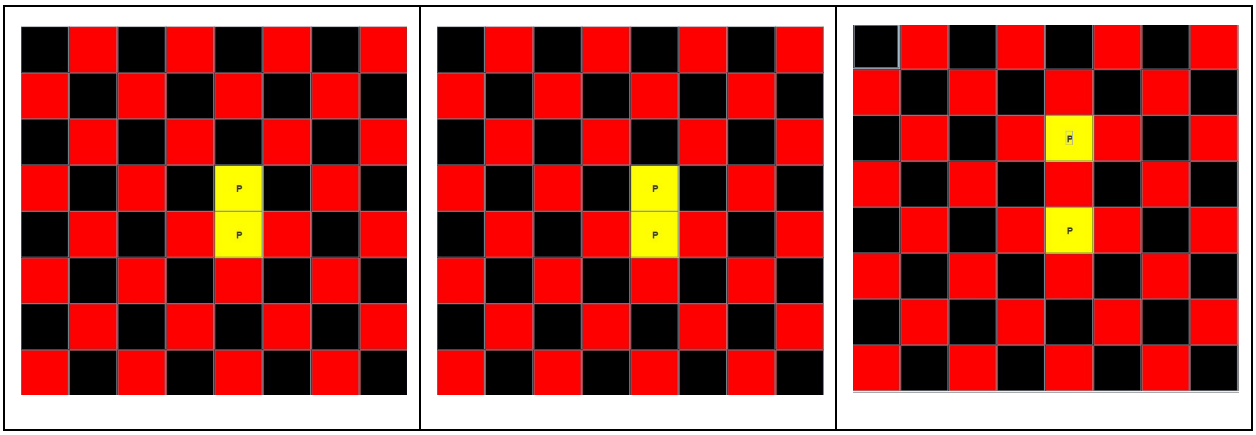
Description: If we try to move the Pawn horizontally, the Pawn will stand still. After the Pawn at position (6, 4) moves two steps, if we try to move another two steps, the game will not allow you to do so because the first move has been made. Therefore, all we can do now is to move one step.



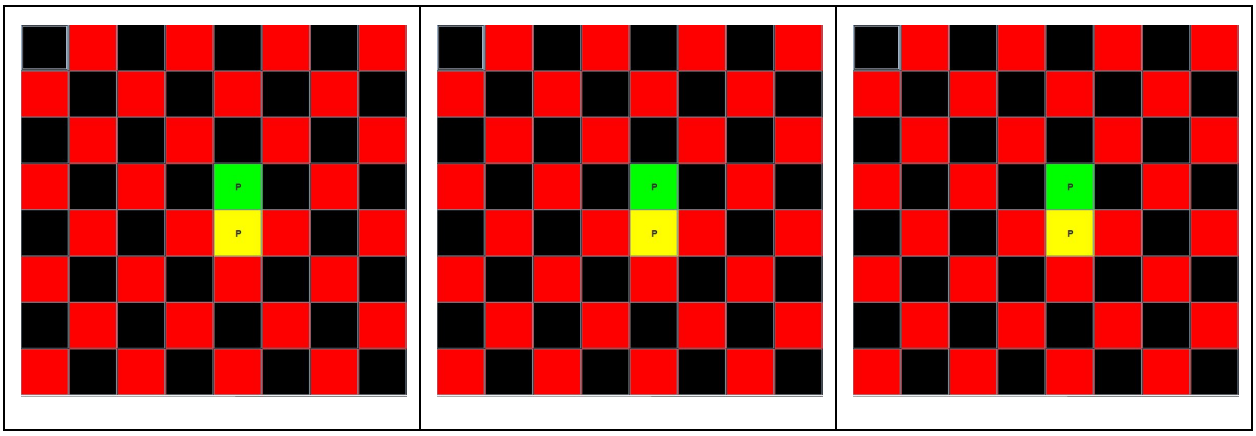
Description: After the Pawn at position (6, 4) moves one step, if we try to move another two steps, the game will not allow you to do so because the first move has been made. Therefore, all we can do now is to move one step.



Description: We can see on the images above that the two pawns of the same side are stacked up. This makes the only move that could be done is to move the higher pawn one step because the yellow pawns can only move up. The pawns cannot move backward!

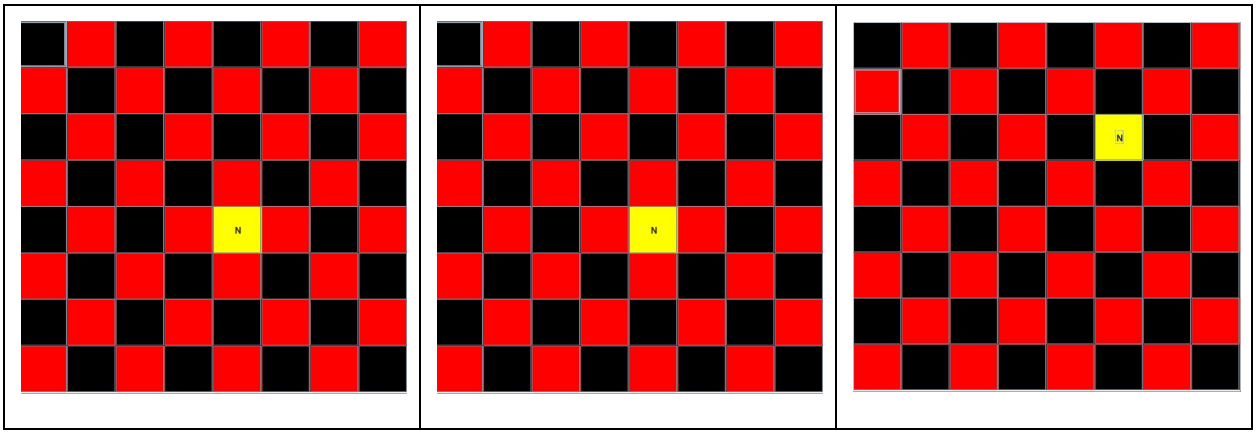


Description: We can see on the images above that the two pawns are stacked up. This tells us that there are no moves that could be done now because pawns can only move up. Whereas green and yellow pawns move in opposite directions, making this impossible.

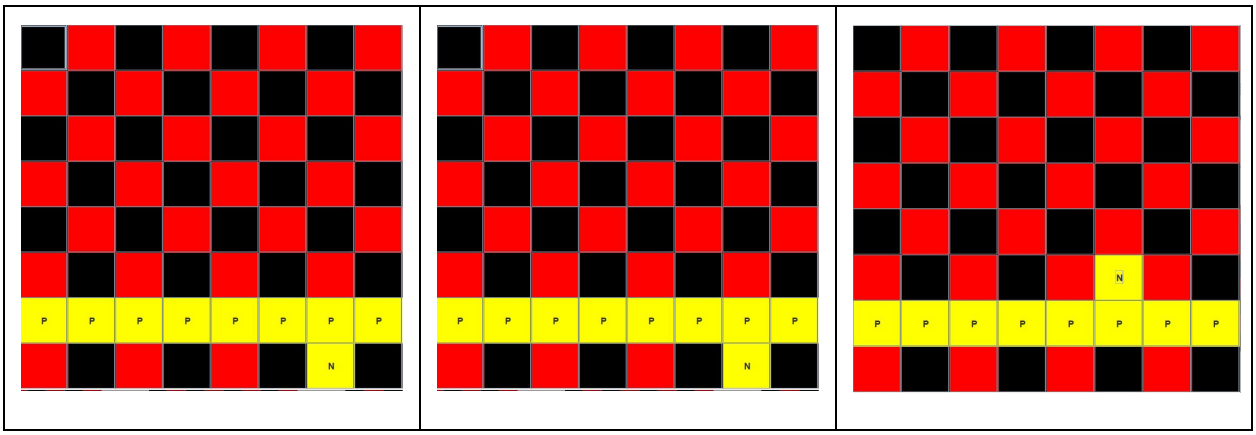


Test 2.2: test KnightPiece move (PASSED)

Description: When we try to move somewhere not in the L-path, the Knight will stand at its own place. However, when we move the Knight from (4, 4) to (2,5), it is a L-path, which is valid.

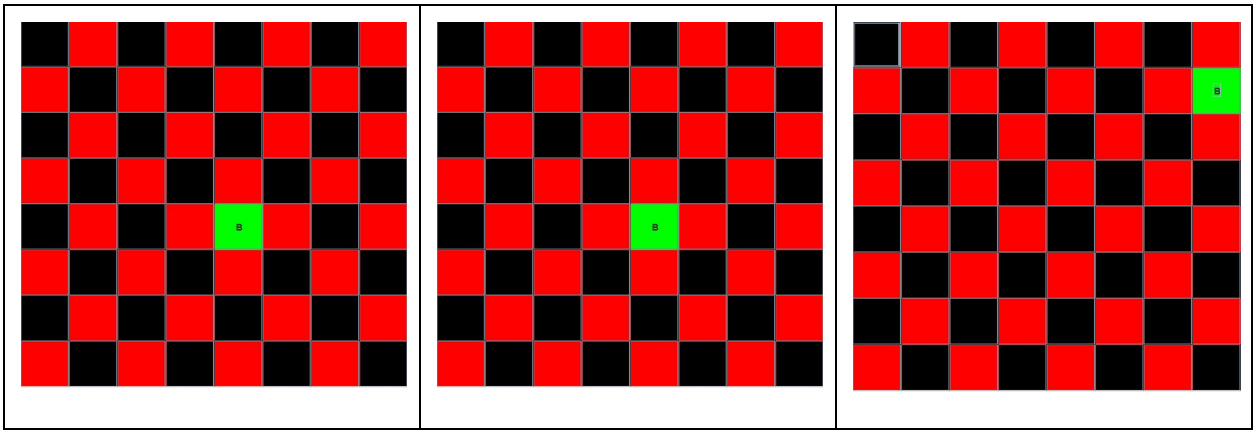


Description: When we try to move somewhere not in the L-path, the Knight will stand at its own place. Also, when we try to travel to any place that has a Pawn of its own side, the Knight will also stand still. However, when we move the Knight from (7, 6) to (5,5), it is a L-path. We also notice that the Knight actually jumps through the Pawn, which makes this test valid.

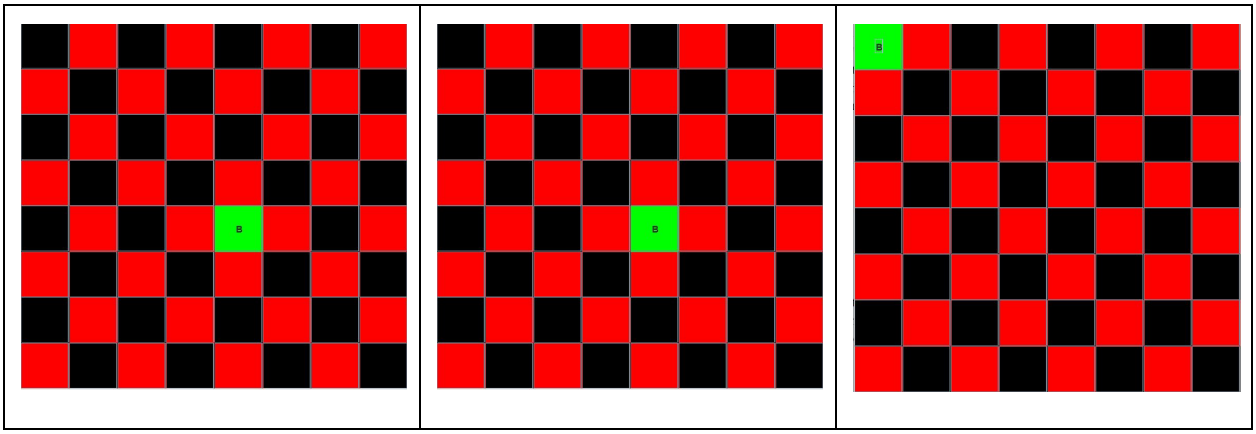


Test 2.3: test BishopPiece move (PASSED)
Note: A piece will move diagonally if abs(delta_Row) = abs(delta_Column)

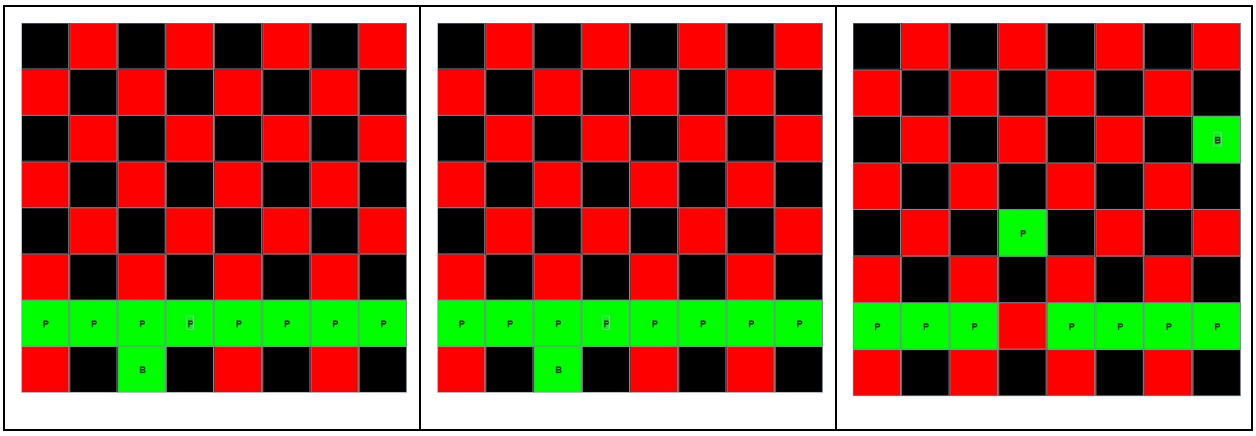
Description: When we try to move horizontally or vertically, the Bishop will stand at its own place. However, when we move the Knight from (4, 4) to (1,7), it is a diagonal path, following the sub-diagonal of the board, which is valid.



Description: When we try to move horizontally or vertically, the Bishop will stand at its own place. However, when we move the Bishop from (4, 4) to (0, 0), it is a diagonal path, following the main-diagonal of the board, which is valid.

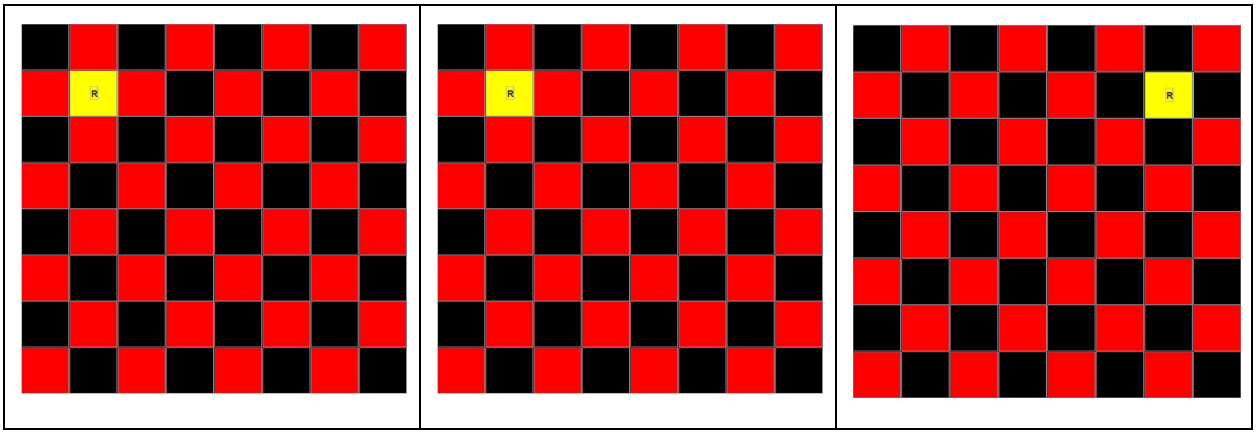


Description: When we try to move diagonally, the Bishop will stand at its own place because it is blocked by a piece of its own side. However, when we move the Pawn from (6, 3) to (4, 3) and then move Bishop from (4, 4) to (0, 0), it is a diagonal path, following the sub-diagonal of the board, which is valid.

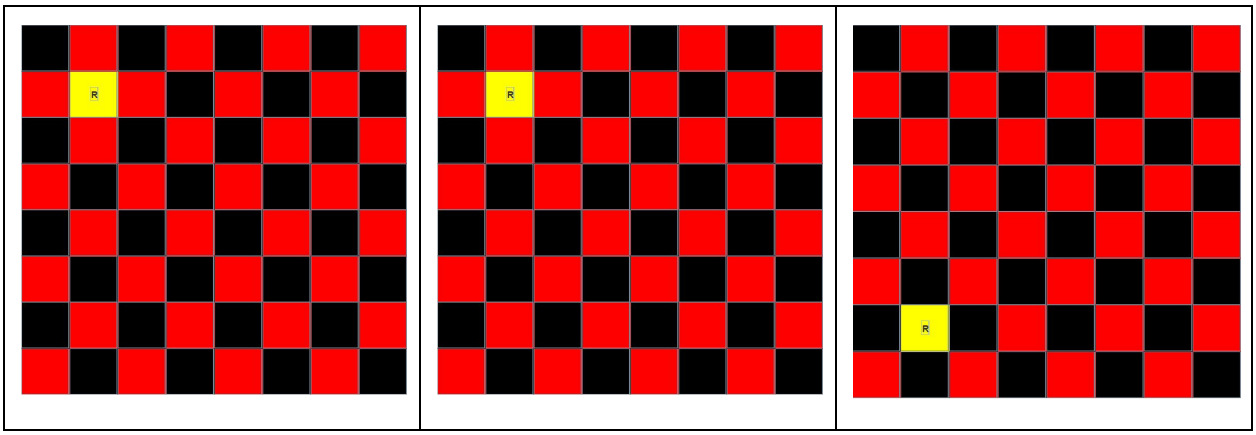


Test 2.4: test RookPiece move (PASSED)

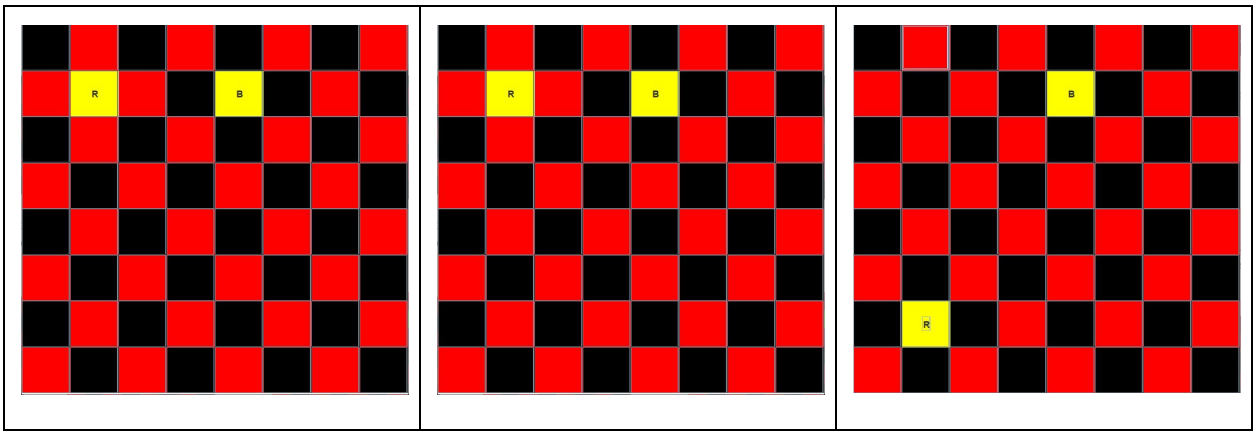
Description: When we try to move diagonally, the Rook will stand at its own place. However, when we move the Bishop from (1, 1) to (1, 6), it is a horizontal path, which is valid.



Description: When we try to move diagonally, the Rook will stand at its own place. However, when we move the Bishop from (1, 1) to (6, 1), it is a vertical path, which is valid.

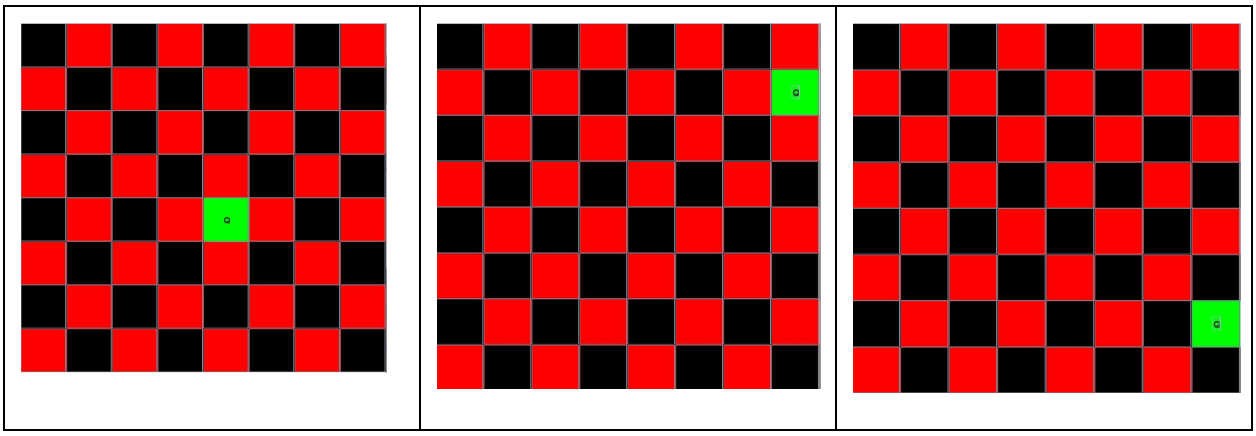


Description: When we try to move to (1, 6), the Rook will stand at its own place because the location the Rook travels to is neither empty nor an opposite side. However, when we move the Bishop from (1, 1) to (6, 1), it is a vertical path, which is valid. Similar for blocked vertical movement.

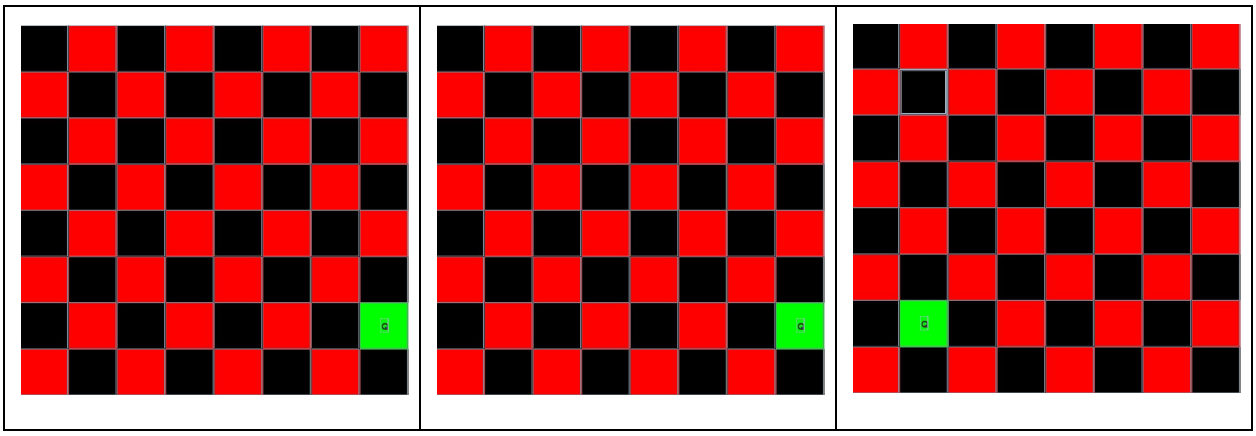


Test 2.5: test QueenPiece move (PASSED)

Description: We will first move the queen diagonally from (4, 4) to (1, 7) and then move vertically from (1, 7) to (6, 7). Both movements are valid.



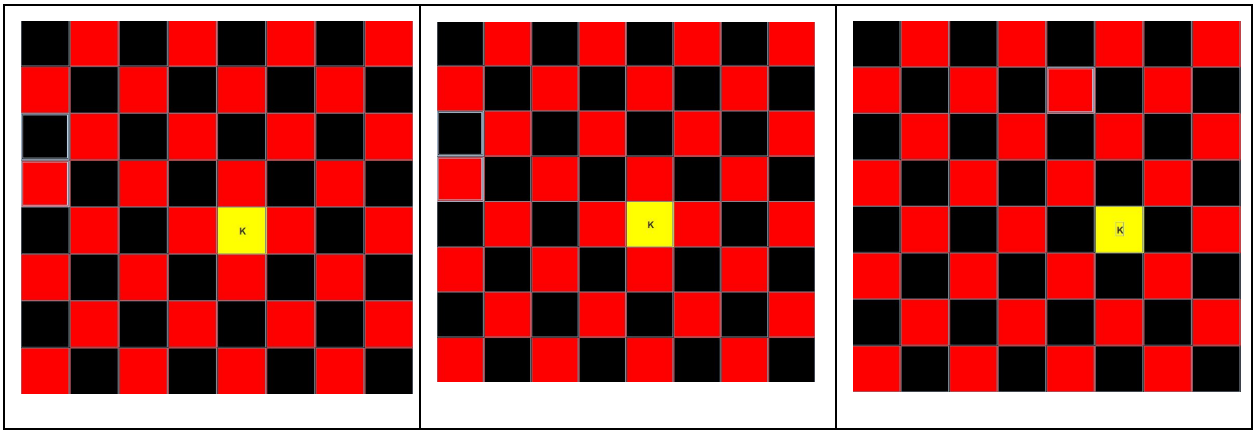
Description: When we try to move the Queen L-path from (6, 7) to (4, 6), the Queen will stand still because the path is invalid. However, when we move horizontally from (6, 7) to (6, 1), the path is valid.



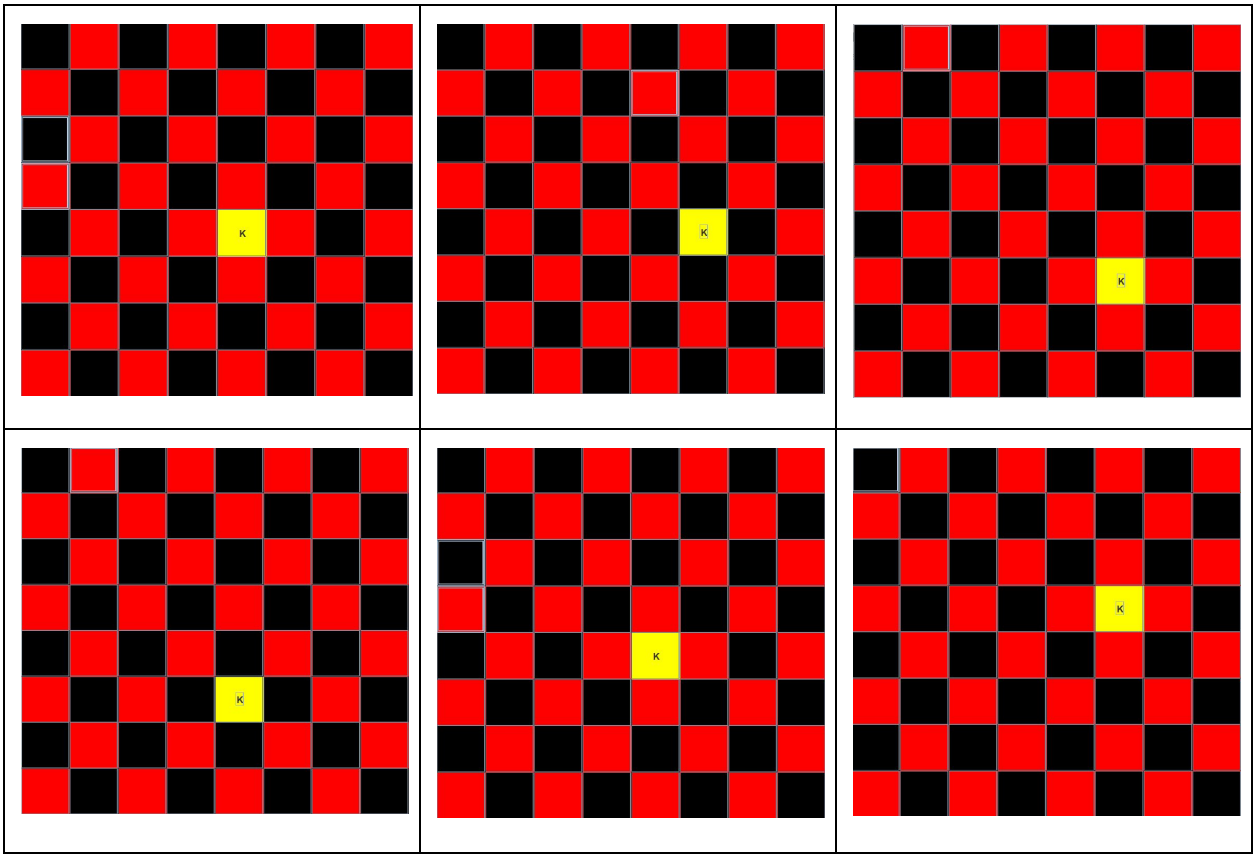
Note: *the Queen moves similar to Rook and Bishop combined. So when there is a piece of its own side on the Queen's path, the movement would be invalid.*

Test 2.6: test KingPiece move (PASSED)

Description: When we try to move the King L-path from (4, 4) to (5, 2), the King will stand still because the path is invalid. Moreover, when we try to move the King 2 steps horizontally or vertically (for example: horizontally from (4, 4) to (4, 6)), the King will stand still because the path is invalid. However, when we move horizontally from (4, 4) to (4, 5), the path is valid horizontally.



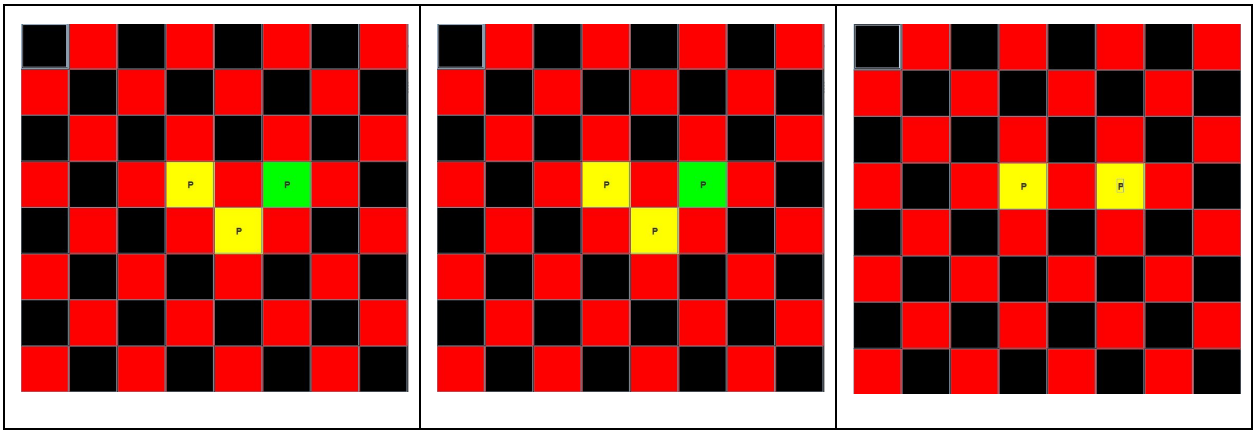
Description: We can move the King one step up, down, left, right or diagonal. Move the King in the following order: right -> down -> left -> up -> diagonal. All directions valid!



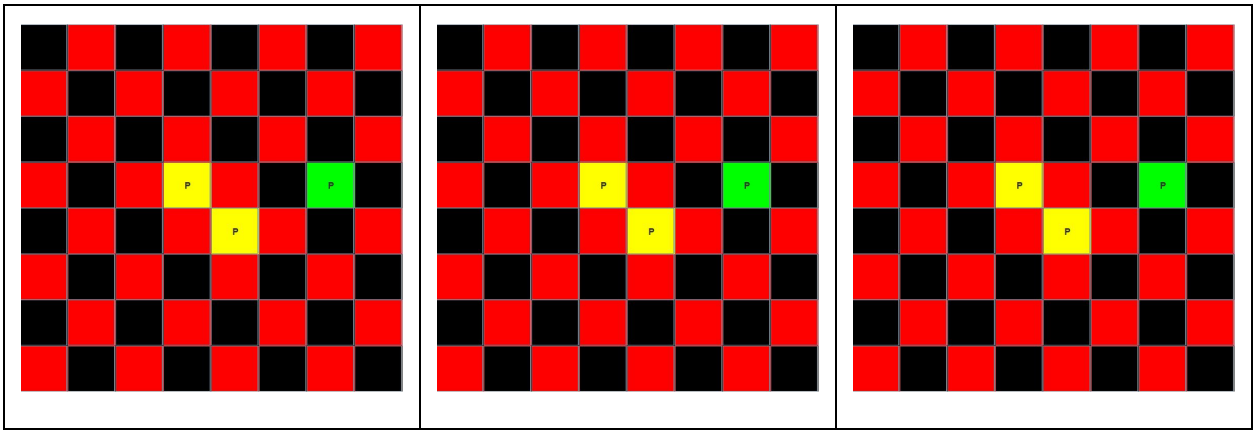
3) TEST EUROPEAN CHESS PIECE CAPTURE

Test 3.1: test PawnPiece capture (PASSED)

Description: We can see that the Pawn at (4, 4) has 2 other pieces in its diagonal paths. Those two pieces are of different sides, one similar to the Pawn at (4, 4) and the other on the other side. We know that Pawn can capture only pieces of the different side. In the first time, we tried to capture the yellow piece, and we failed because the two are on the same side. In the second time, we captured the green piece, which is valid since it is on the different side and the movement of the Pawn is valid.

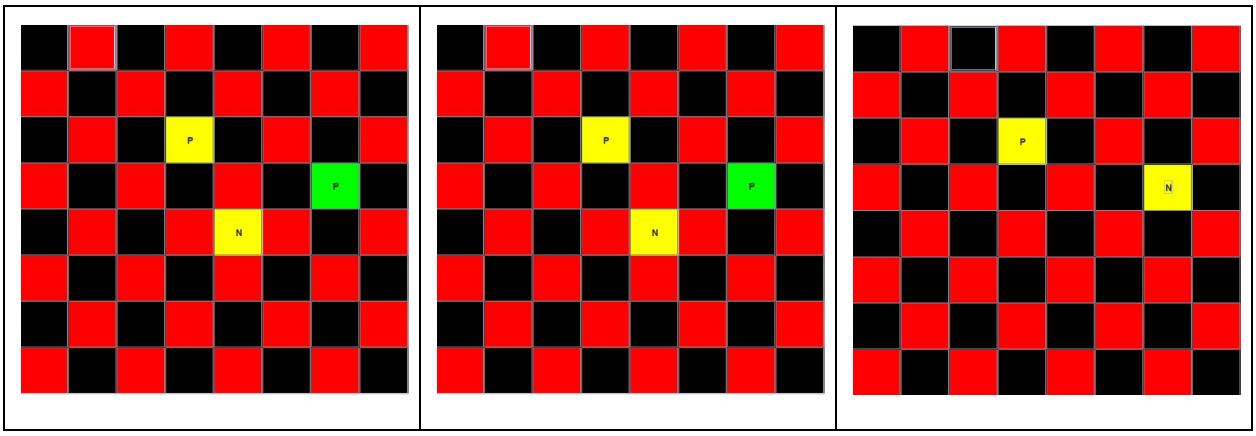


Description: The opponent piece is not valid to capture because is it not legal to move the Pawn of place (4, 4) in one move to reach it. Both possible capture places are either the place of the piece on the same side or empty space.

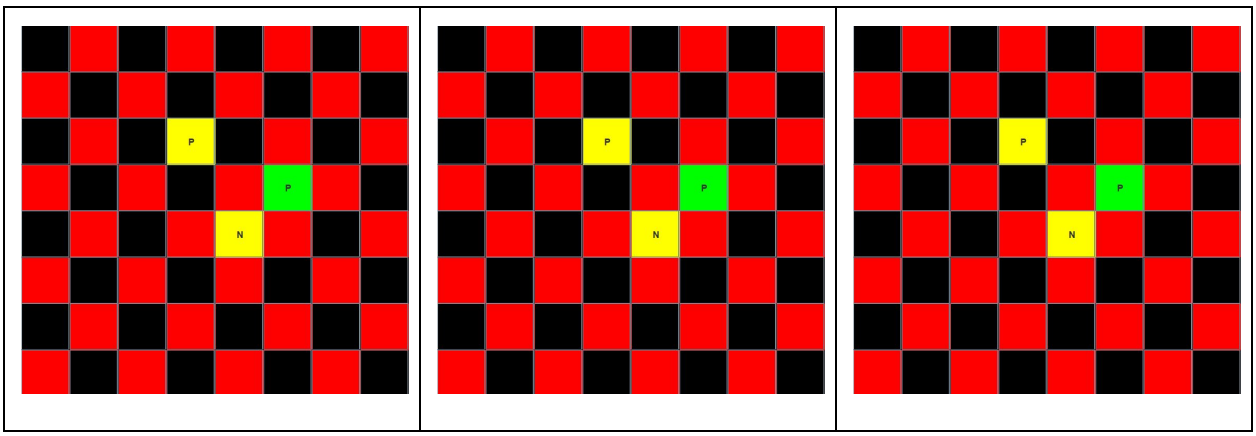


Test 3.2: test KnightPiece capture (PASSED)

Description: We can see that the Knight at (4, 4) has two pieces in its L-type paths. Those two pieces are of different sides, one similar to the Pawn at (4, 4) and the other on the other side. We know that Knight can capture only pieces of the different side. In the first time, we tried to capture the yellow piece, and we failed because the two are on the same side. In the second time, we captured the green piece, which is valid since it is on the different side and the movement of the Knight is valid.

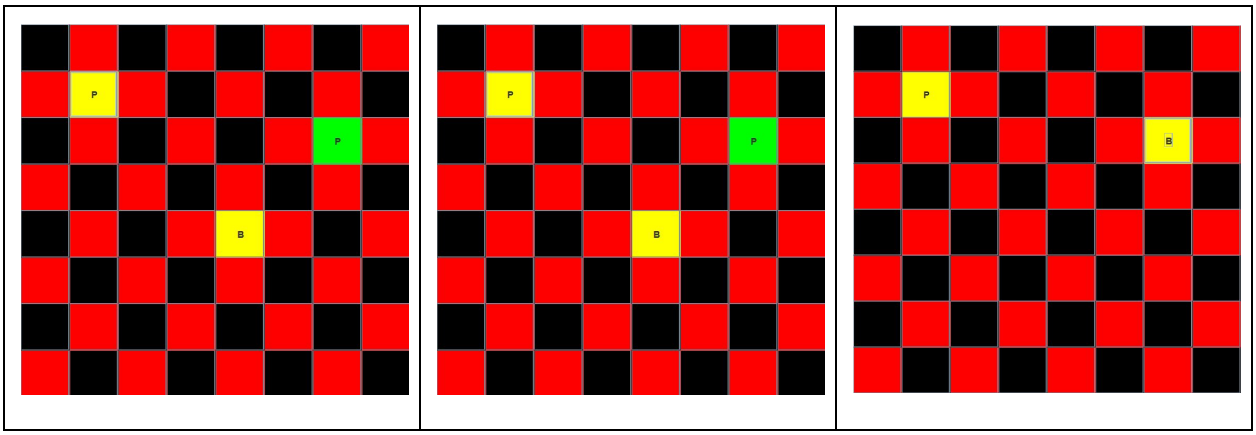


Description: The opponent piece is not valid to capture because is it not legal to move the Knight of place (4, 4) in one move to reach it. Both possible capture places are either the place of the piece on the same side or empty space.

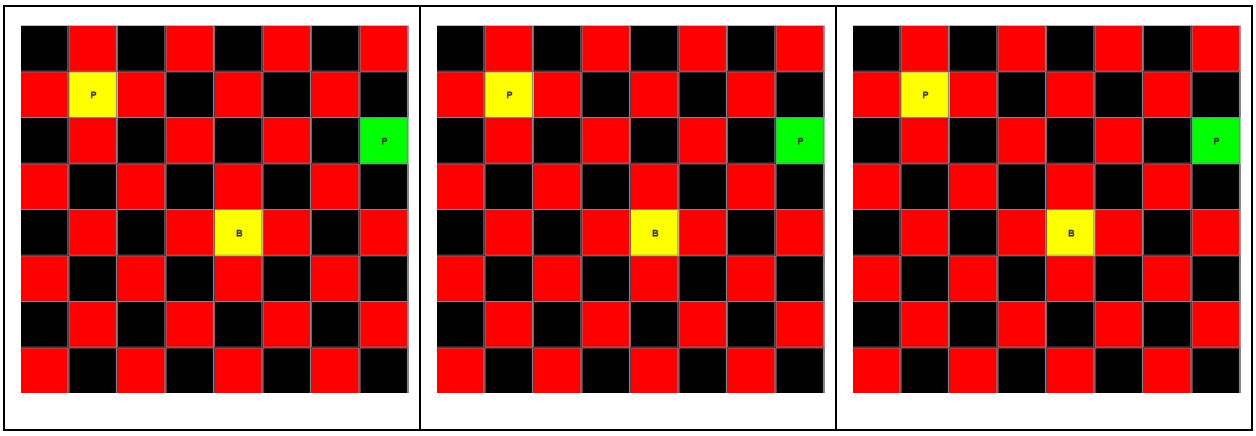


Test 3.3: test BishopPiece capture (PASSED)

Description: We can see that the Bishop at (4, 4) has two pieces in its diagonal paths. Those two pieces are of different sides, one similar to the Bishop at (4, 4) and the other on the other side. We know that Bishop can capture only pieces of the different side. In the first time, we tried to capture the yellow piece, and we failed because the two are on the same side. In the second time, we captured the green piece, which is valid since it is on the different side and the movement of the Bishop is valid.

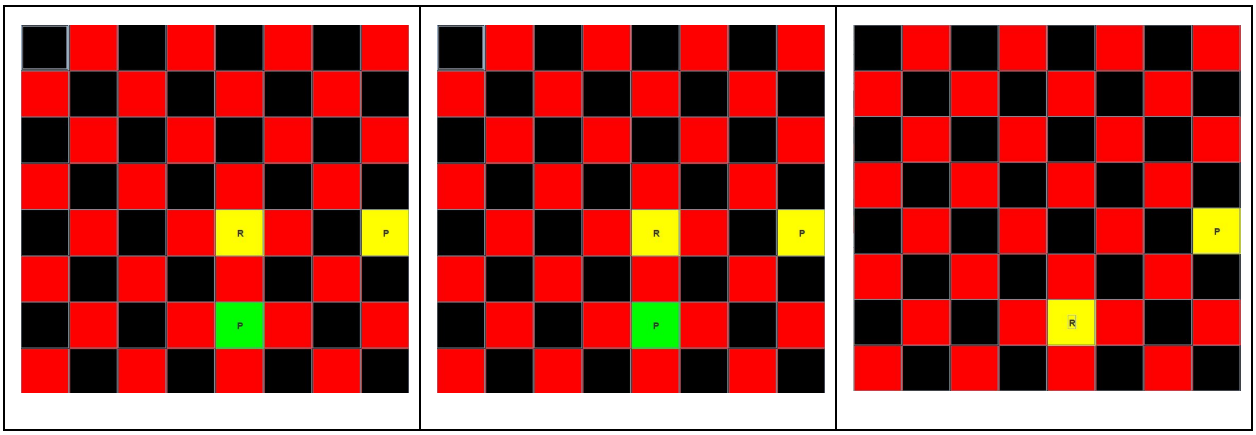


Description: The opponent piece is not valid to capture because is it not legal to move the Knight of place (4, 4) in one move to reach it. Both possible capture places are either the place of the piece on the same side or empty space.

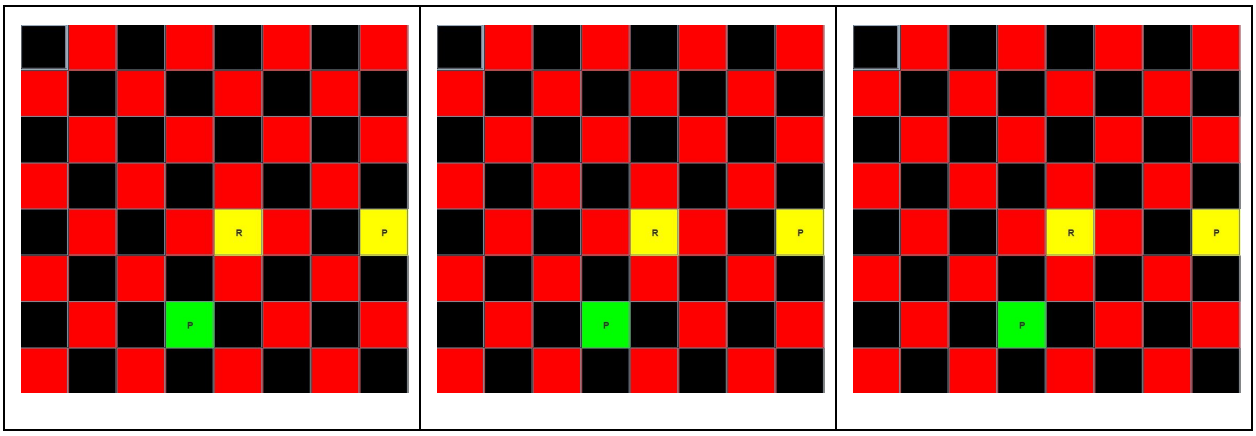


Test 3.4: test RookPiece capture (PASSED)

Description: We can see that the Rook at (4, 4) has two pieces in its straight paths. Those two pieces are of different sides, one similar to the Rook at (4, 4) and the other on the other side. We know that Rook can capture only pieces of the different side. In the first time, we tried to capture the yellow piece, and we failed because the two are on the same side. In the second time, we captured the green piece, which is valid since it is on the different side and the movement of the Rook is valid.

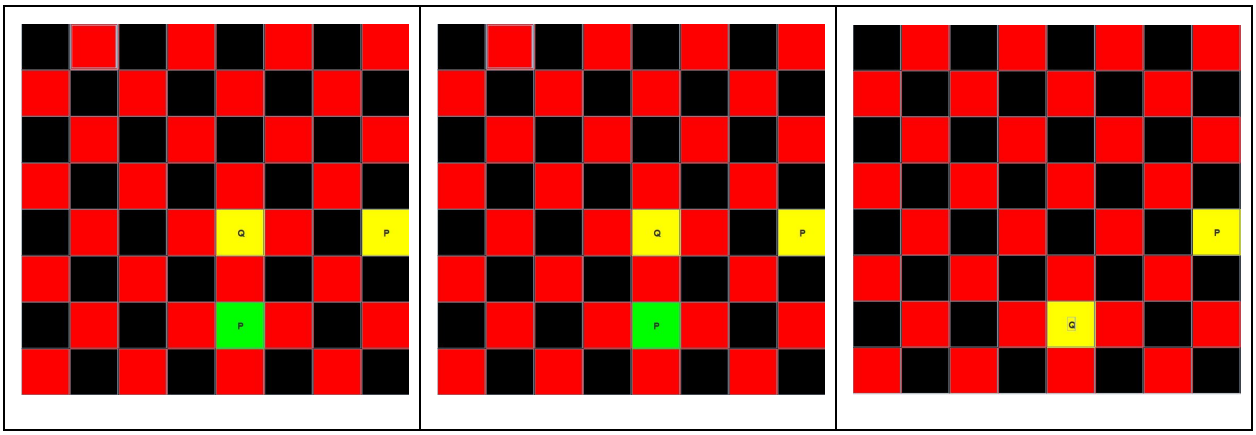


Description: The opponent piece is not valid to capture because is it not legal to move the Rook of place (4, 4) in one move to reach it. Both possible capture places are either the place of the piece on the same side or empty space.

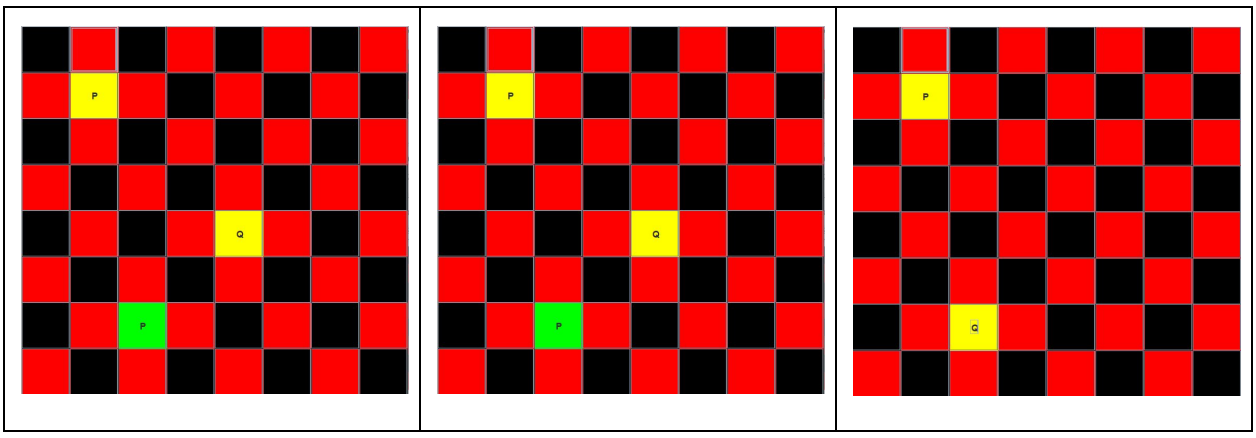


Test 3.5: test QueenPiece capture (PASSED)

Description: We can see that the Queen at (4, 4) has two pieces in its straight paths. Those two pieces are of different sides, one similar to the Queen at (4, 4) and the other on the other side. We know that the Queen can capture only pieces of the different side. In the first time, we tried to capture the yellow piece, and we failed because the two are on the same side. In the second time, we captured the green piece, which is valid since it is on the different side and the movement of the Queen is valid.

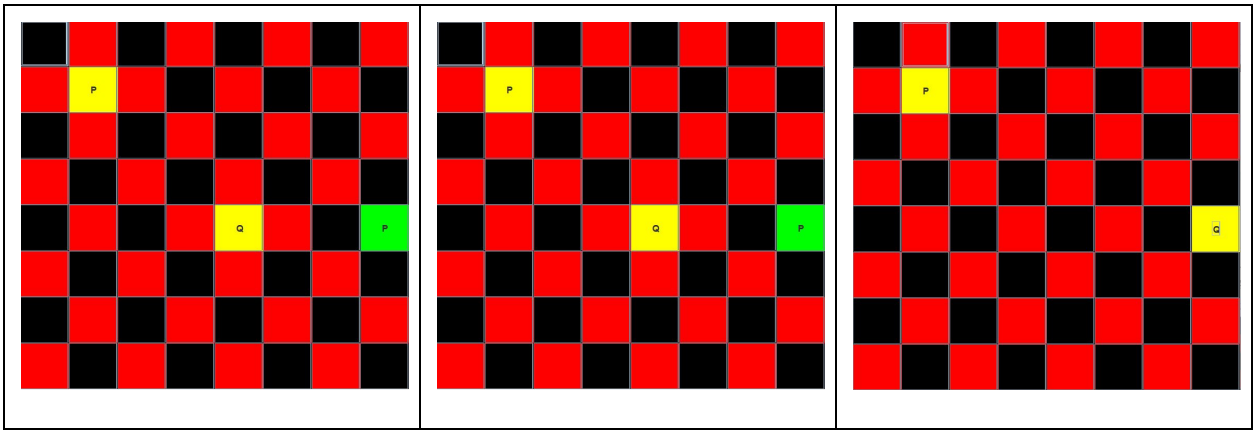


Description: We can see that the Queen at (4, 4) has two pieces in its diagonal paths. Those two pieces are of different sides, one similar to the Queen at (4, 4) and the other on the other side. We know that the Queen can capture only pieces of the different side. In the first time, we tried to capture the yellow piece, and we failed because the two are on the same side. In the second time, we captured the green piece, which is valid since it is on the different side and the movement of the Queen is valid.

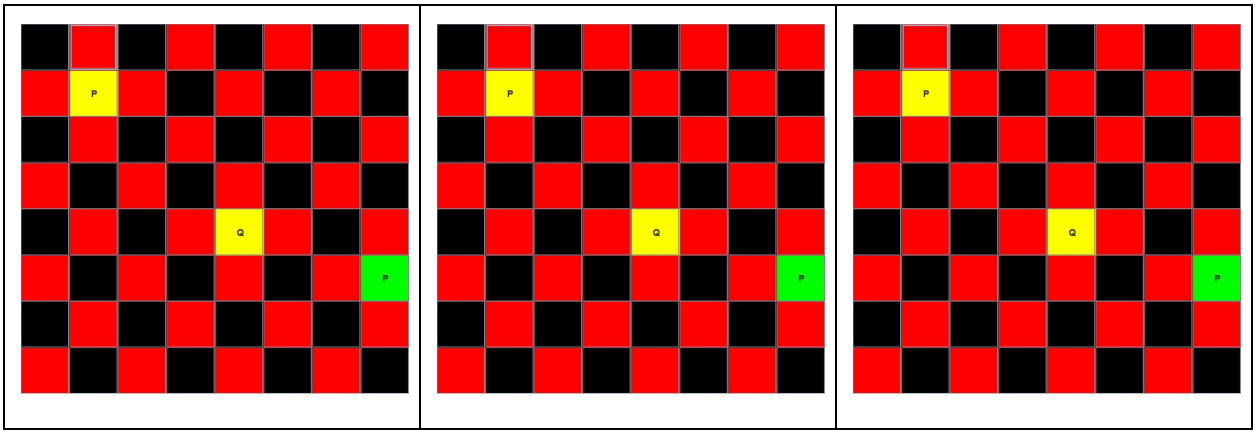


Description: We can see that the Queen at (4, 4) has one piece in its straight path and one piece in its diagonal path. Those two pieces are of different sides, one similar to the

Queen at (4, 4) and the other on the other side. We know that the Queen can capture only pieces of the different side. In the first time, we tried to capture the yellow piece, and we failed because the two are on the same side. In the second time, we captured the green piece, which is valid since it is on the different side and the movement of the Queen is valid.

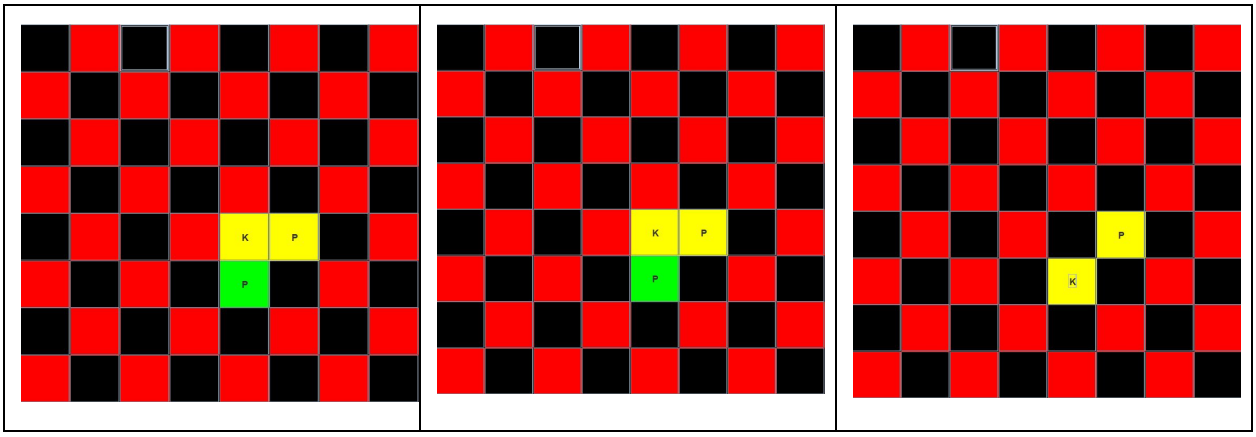


Description: The opponent piece is not valid to capture because is it not legal to move the Queen of place (4, 4) in one move to reach it. Both possible capture places are either the place of the piece on the same side or empty space.

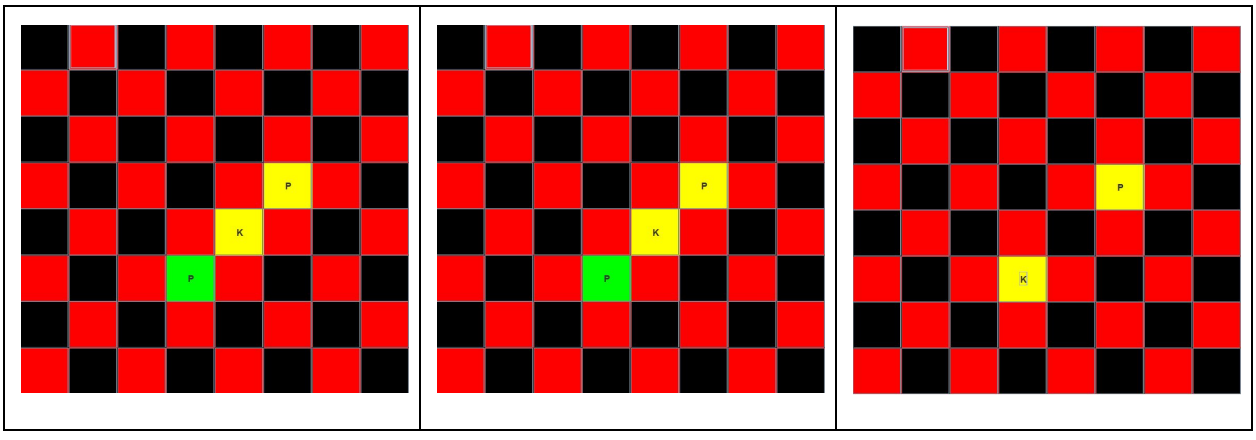


Test 3.6: test KingPiece capture (PASSED)

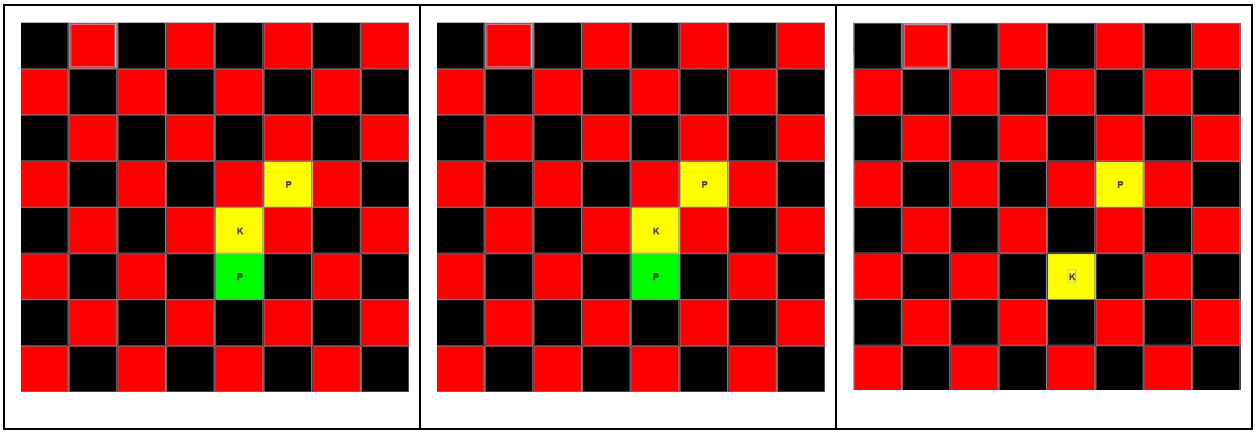
Description: We can see that the King at (4, 4) has two pieces in its straight paths. Those two pieces are of different sides, one similar to the King at (4, 4) and the other on the other side. We know that the King can capture only pieces of the different side. In the first time, we tried to capture the yellow piece, and we failed because the two are on the same side. In the second time, we captured the green piece, which is valid since it is on the different side and the movement of the King is valid.



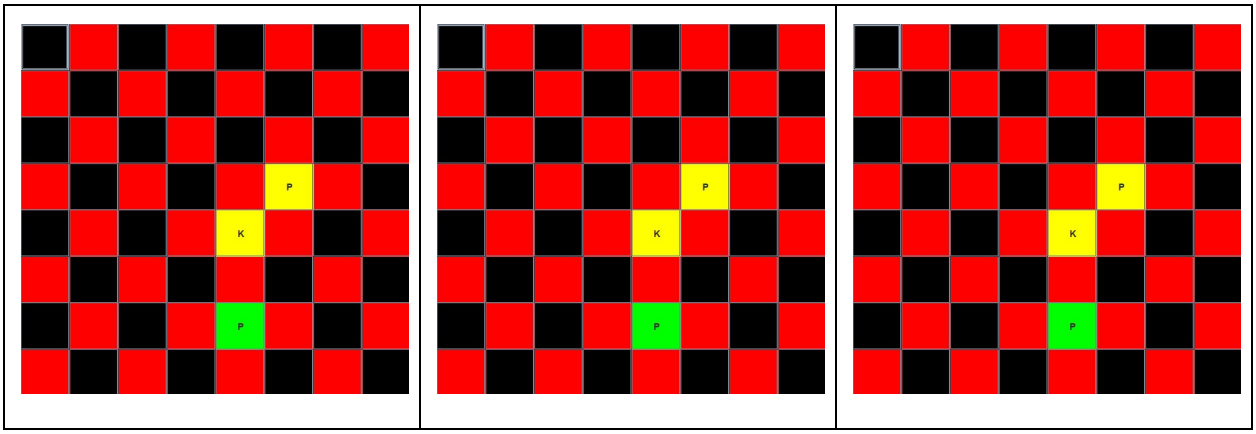
Description: We can see that the King at (4, 4) has two pieces in its diagonal paths. Those two pieces are of different sides, one similar to the King at (4, 4) and the other on the other side. We know that the King can capture only pieces of the different side. In the first time, we tried to capture the yellow piece, and we failed because the two are on the same side. In the second time, we captured the green piece, which is valid since it is on the different side and the movement of the King is valid.



Description: We can see that the King at (4, 4) has one piece in its straight path and one piece in its diagonal path. Those two pieces are of different sides, one similar to the King at (4, 4) and the other on the other side. We know that the King can capture only pieces of the different side. In the first time, we tried to capture the yellow piece, and we failed because the two are on the same side. In the second time, we captured the green piece, which is valid since it is on the different side and the movement of the King is valid.



Description: The opponent piece is not valid to capture because is it not legal to move the King of place (4, 4) in one move to reach it. Both possible capture places are either the place of the piece on the same side or empty space.

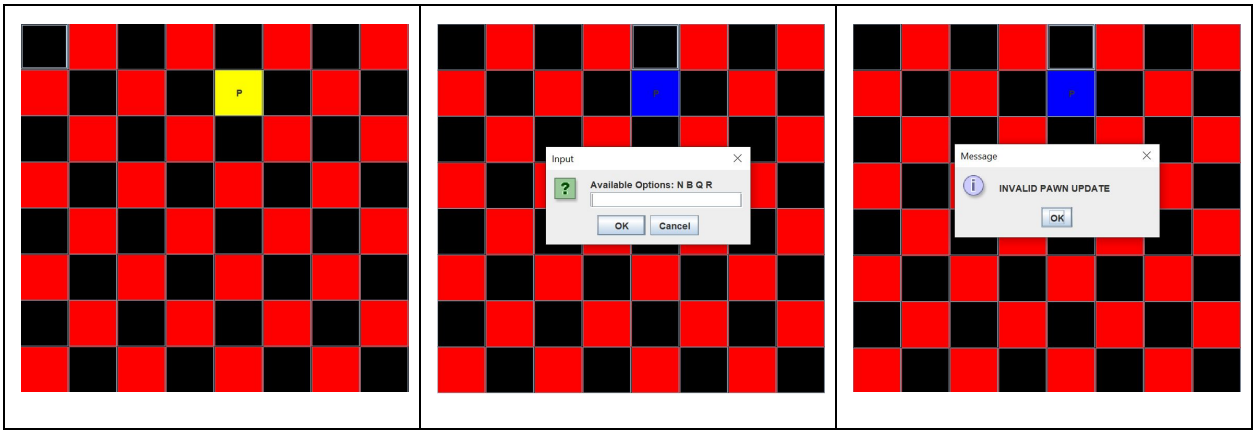


4) TEST PAWN UPDATE

Test 4.1: South Pawn reaches row 0

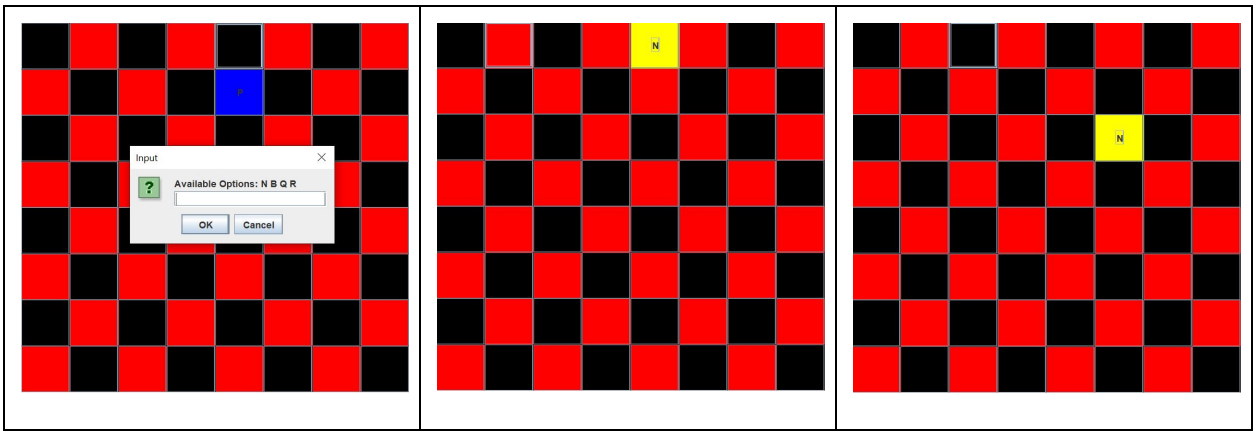
Description: When South Pawn reaches row 0, the game will ask the player which of the pieces to replace. We will have one chance to update the Pawn.

If we don't put anything, put something that is not 'N', 'B', 'Q', 'R', or click cancel, the Pawn will not be updated and a message will pop up saying "INVALID PAWN UPDATE". Finally, it will put a Pawn as if it was moving in that direction without updating.

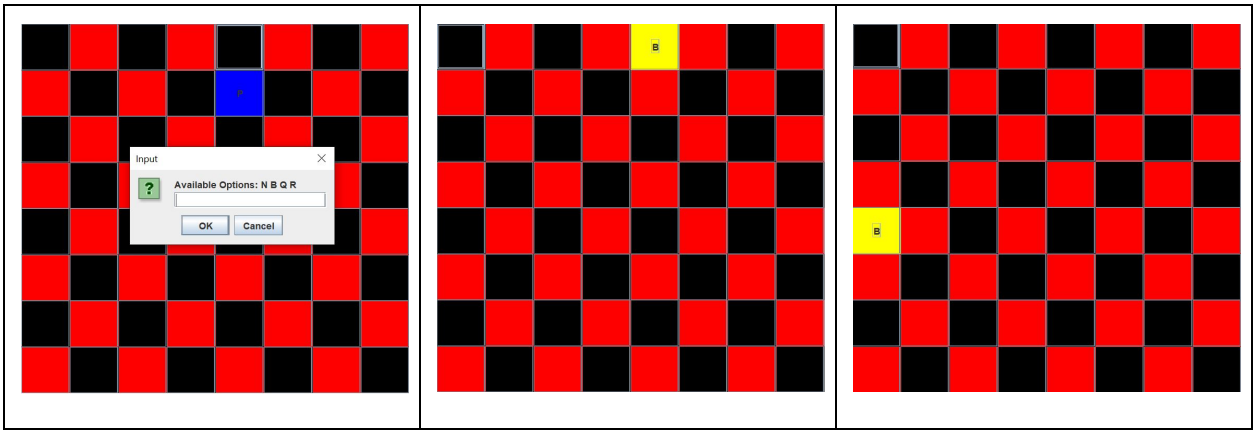


If we put in 'N', 'B', 'Q', or 'R' we will have the corresponding piece added to that position. We will also show that those pieces are functional by making one move.

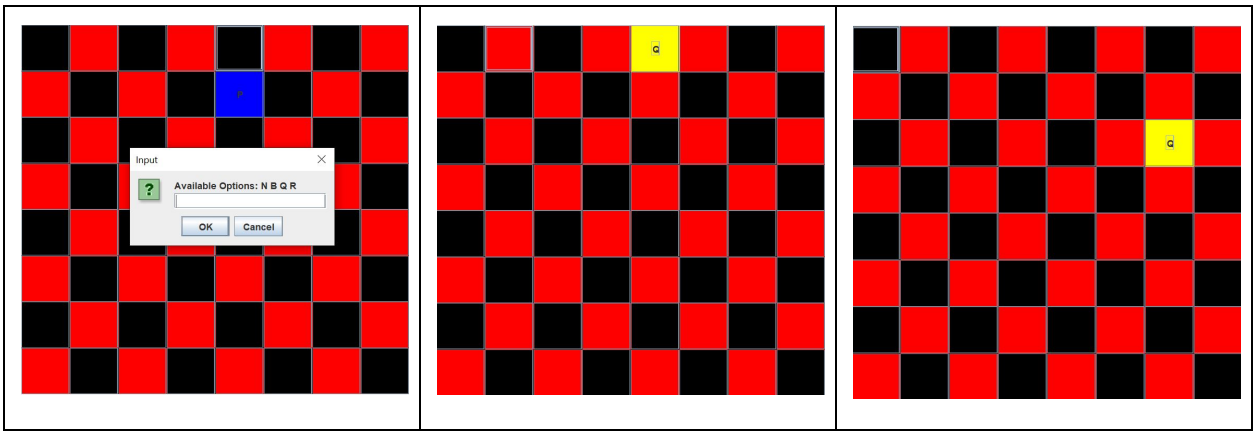
Pawn to Knight



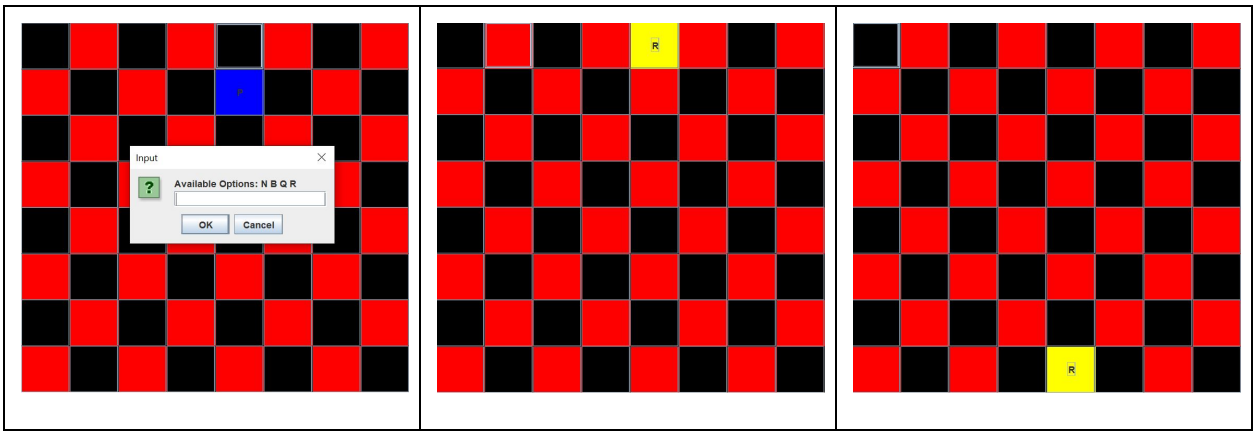
Pawn to Bishop



Pawn to Queen



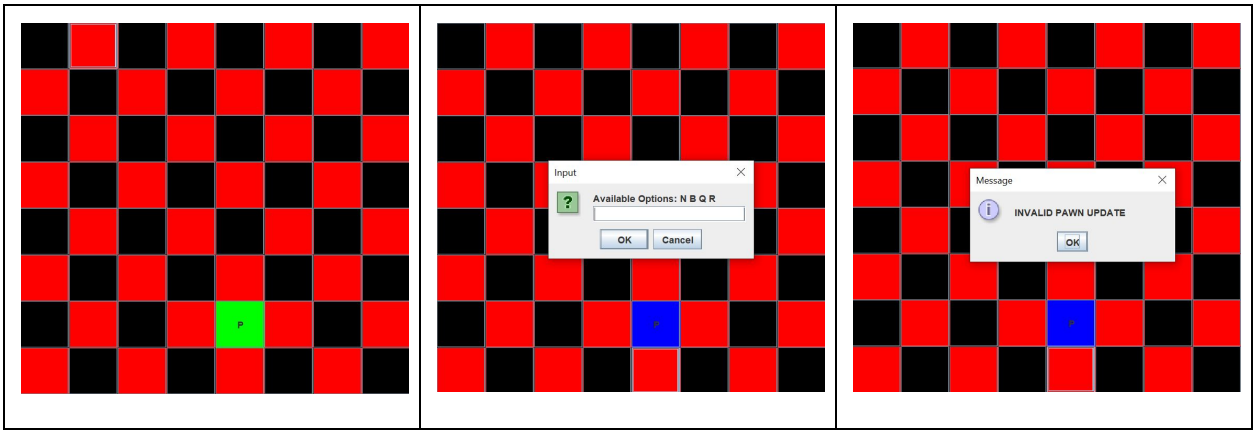
Pawn to Rook



Test 4.2: North Pawn reaches row 7

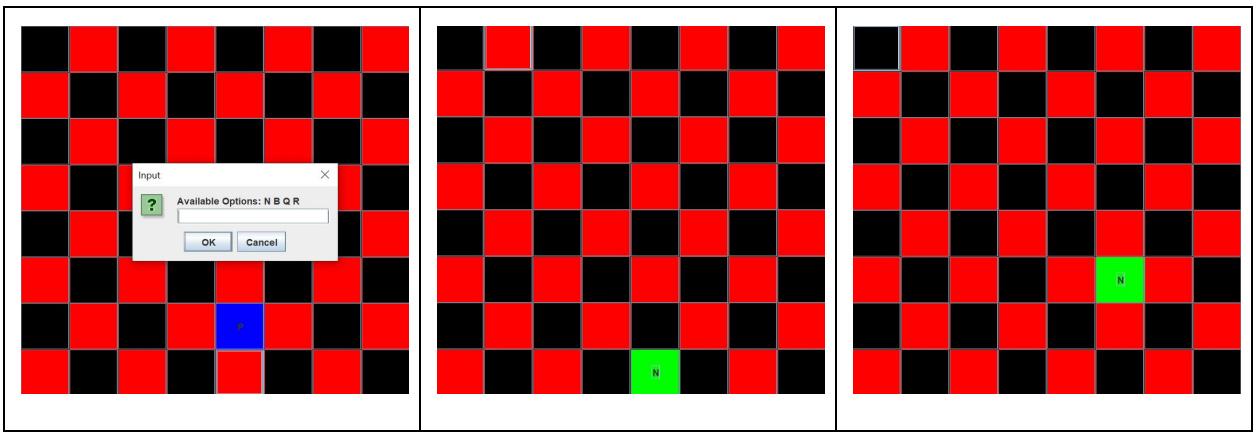
Description: When North Pawn reaches row 7, the game will ask the player which of the pieces to replace. We will have one chance to update the Pawn.

If we don't put anything, put something that is not 'N', 'B', 'Q', 'R', or click cancel, the Pawn will not be updated and a message will pop up saying "INVALID PAWN UPDATE". Finally, it will put a Pawn as if it was moving in that direction without updating.

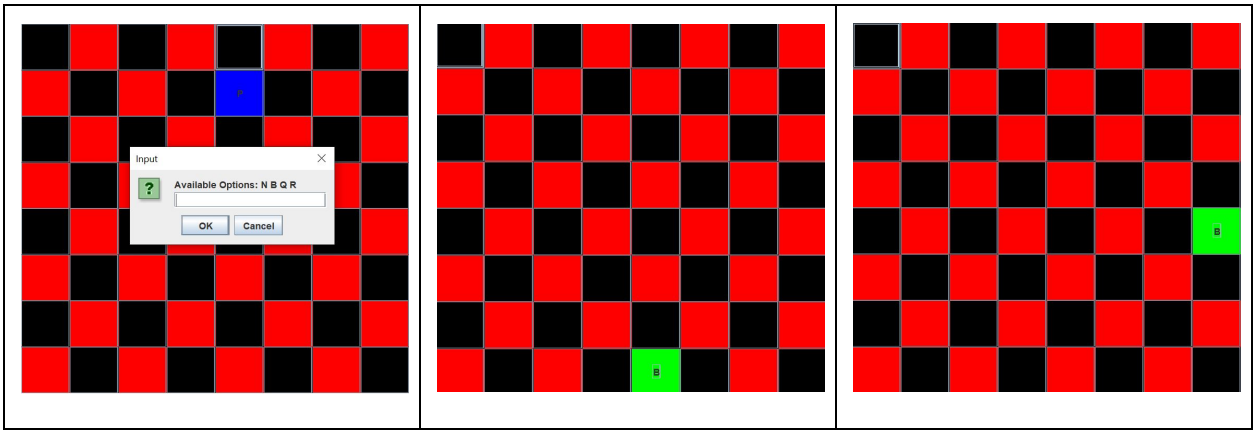


If we put in 'N', 'B', 'Q', or 'R' we will have the corresponding piece added to that position. We will also show that those pieces are functional by making one move.

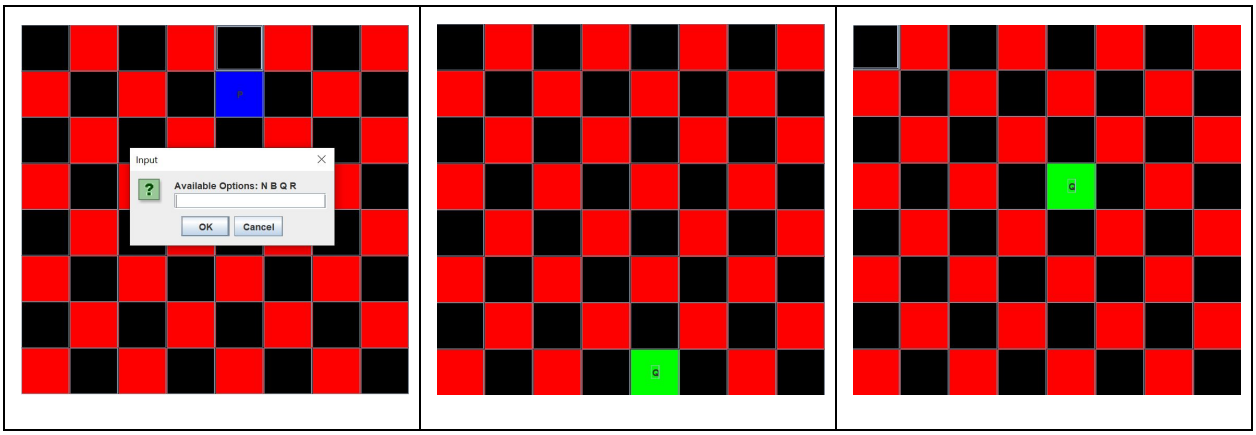
Pawn to Knight



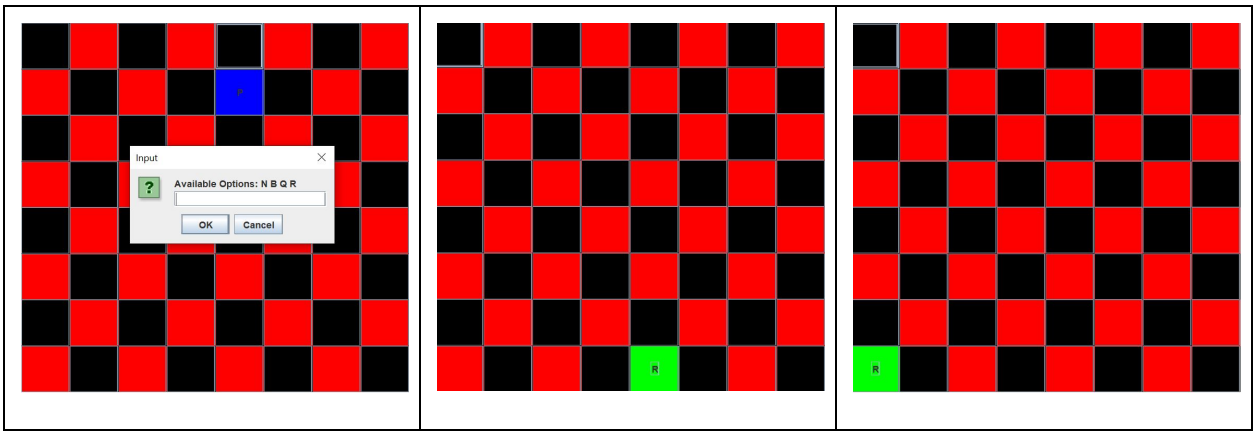
Pawn to Bishop



Pawn to Queen



Pawn to Rook

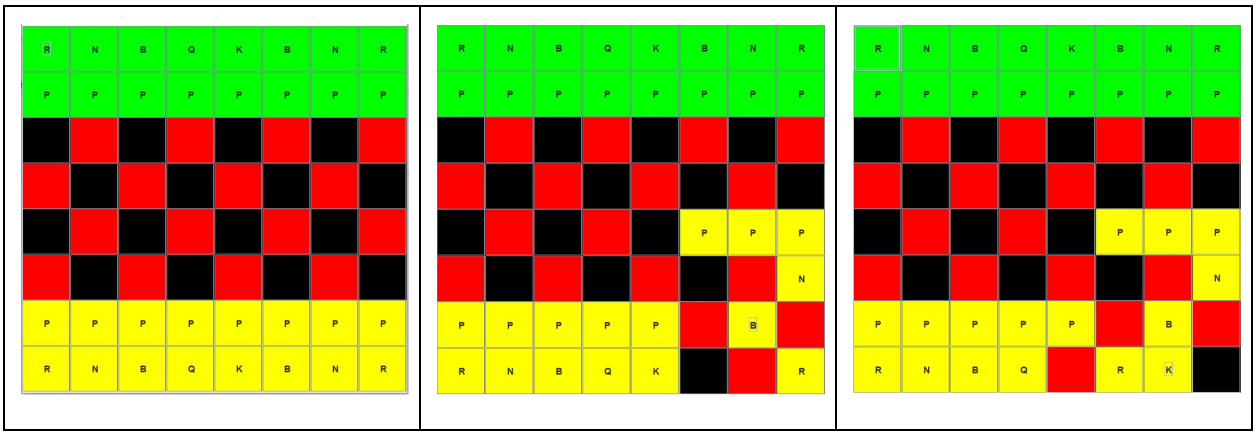


5) TEST CASTLE MOVEMENT

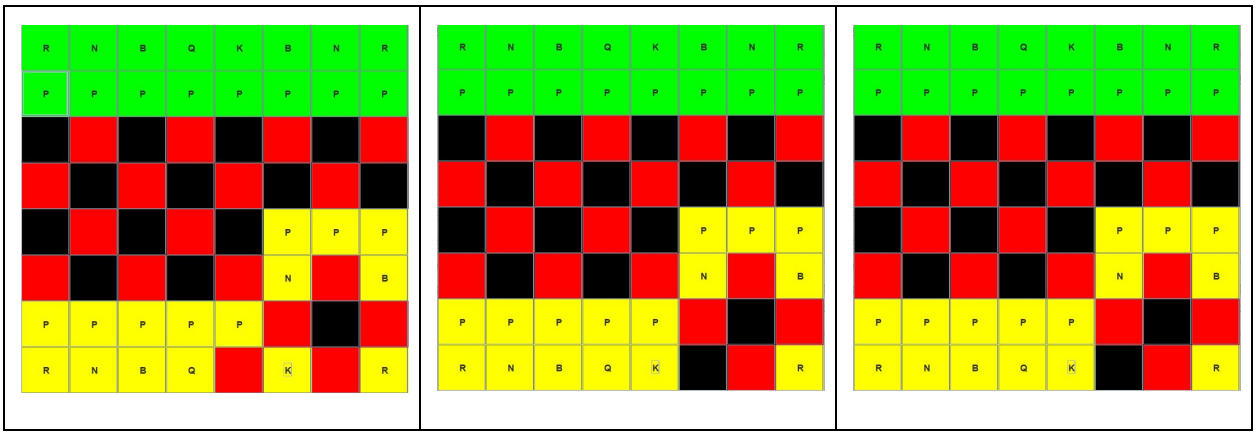
Note: The Castle will only be done if the King and the Rook have never moved. Also, there should be no pieces on the path.

Test 5.1: South Right (PASSED)

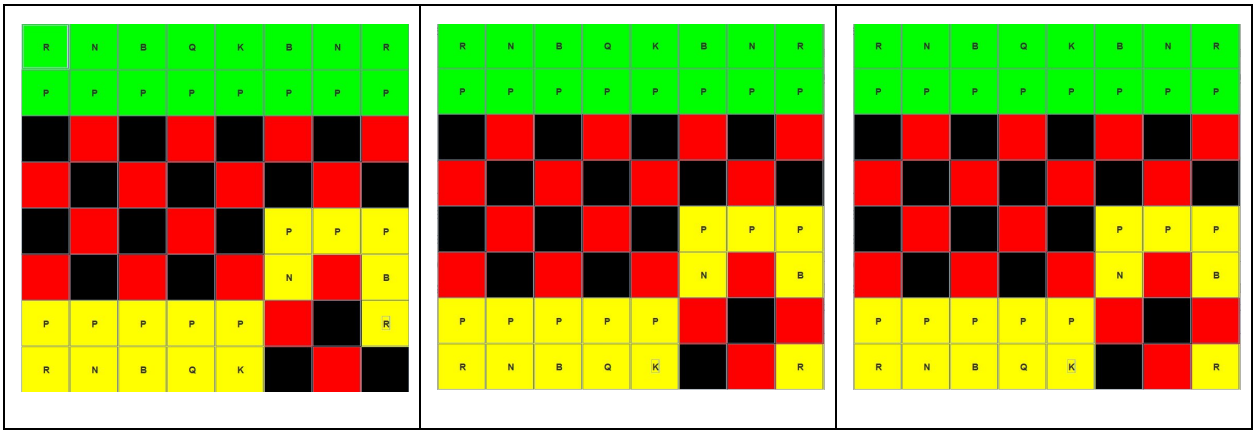
Description: We will set up the table so that this time it is possible for the Castle to move because all conditions are satisfied.



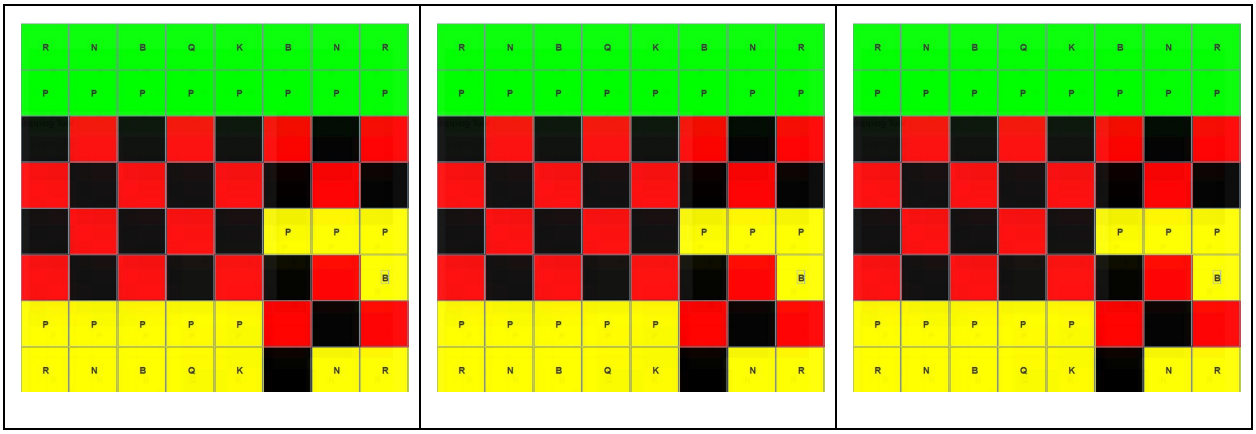
Description: We will set up the table so that this time it is not possible for the Castle to move because King has moved once.



Description: We will set up the table so that this time it is not possible for the Castle to move because Rook has moved once.

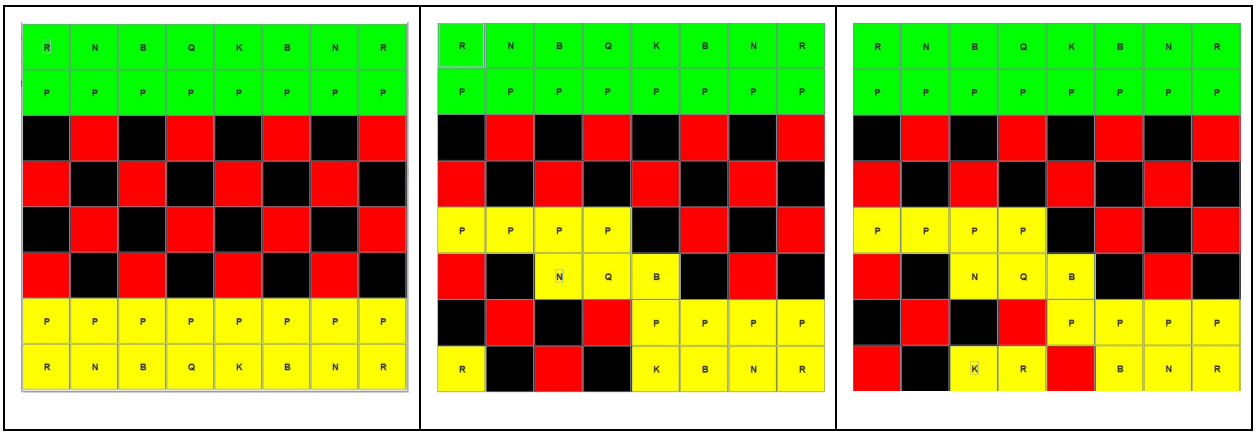


Description: We will set up the table so that this time it is not possible for the Castle to move because there is a block in the path.

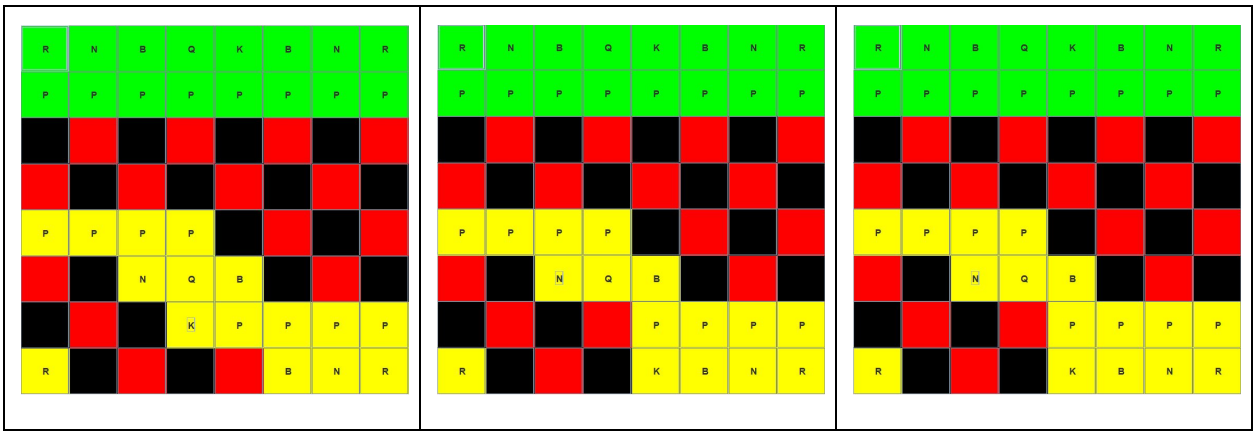


Test 5.2: South Left (PASSED)

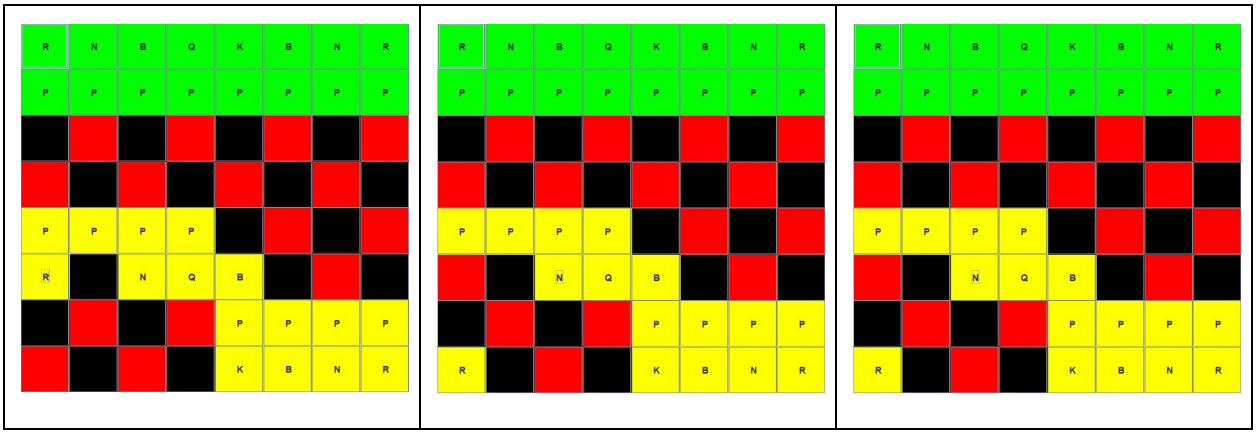
Description: We will set up the table so that this time it is possible for the Castle to move because all conditions are satisfied.



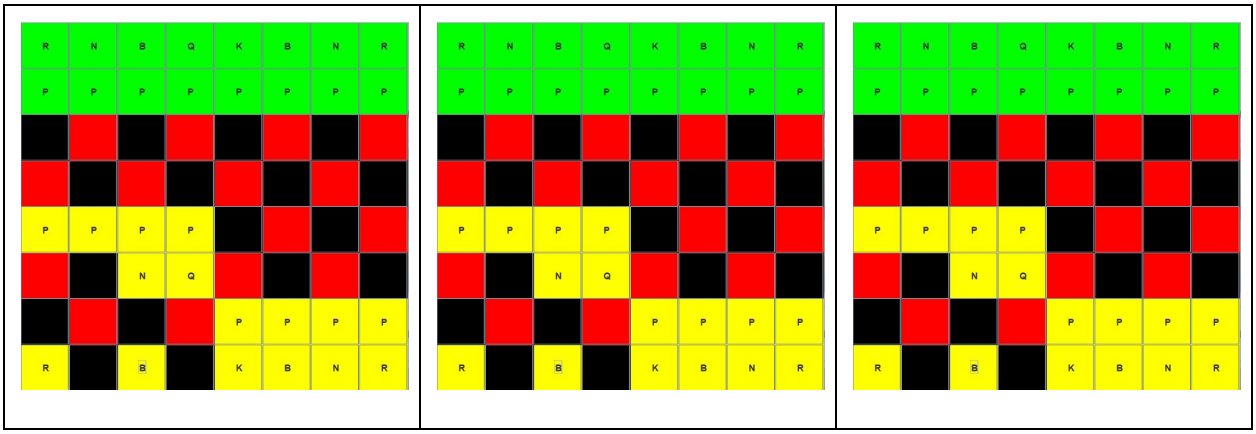
Description: We will set up the table so that this time it is not possible for the Castle to move because King has moved once.



Description: We will set up the table so that this time it is not possible for the Castle to move because Rook has moved once.

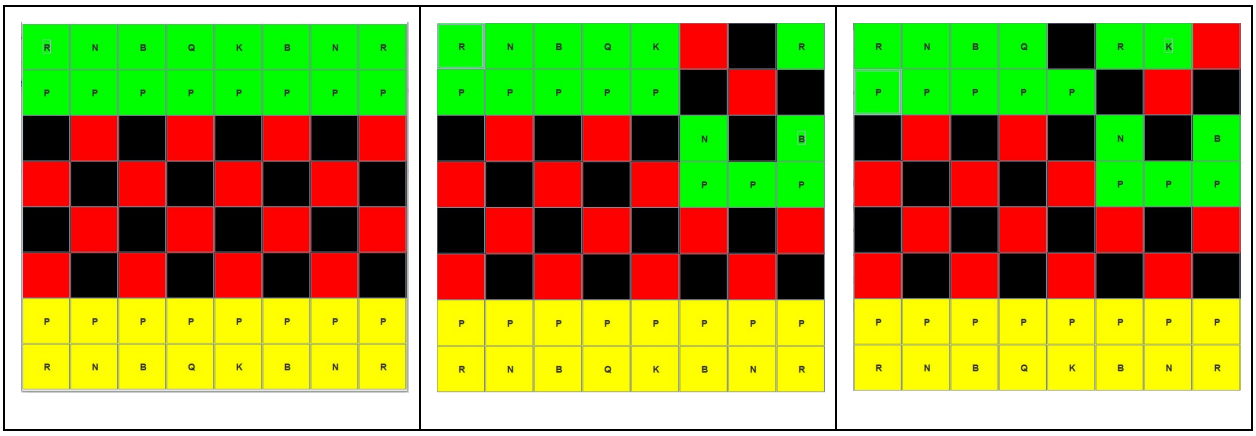


Description: We will set up the table so that this time it is not possible for the Castle to move because there is a block in the path.

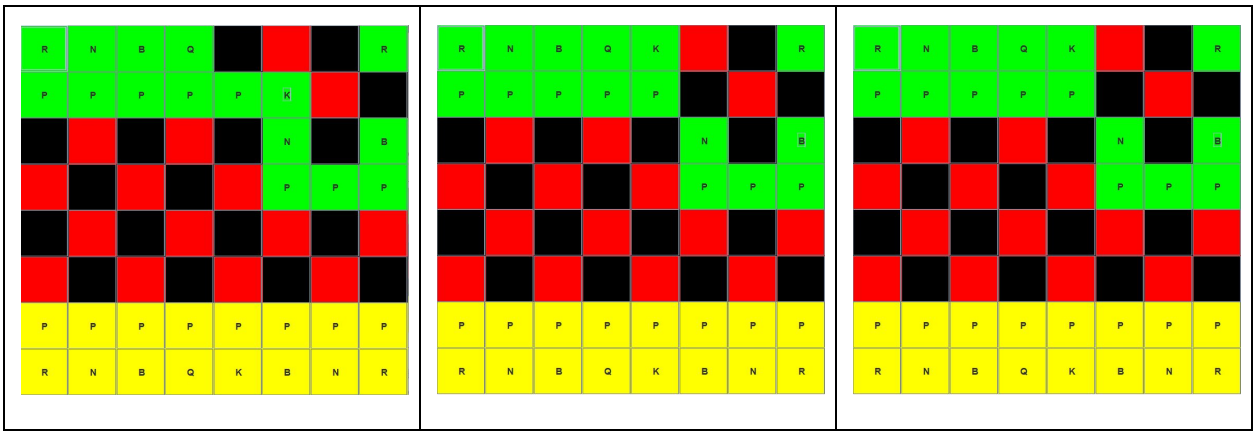


Test 5.3: North Right

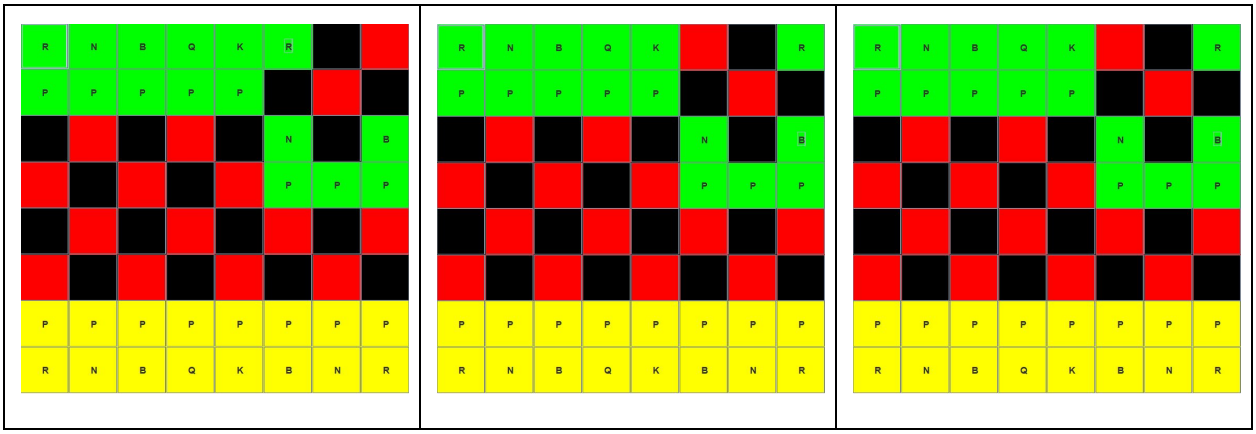
Description: We will set up the table so that this time it is possible for the Castle to move because all conditions are satisfied.



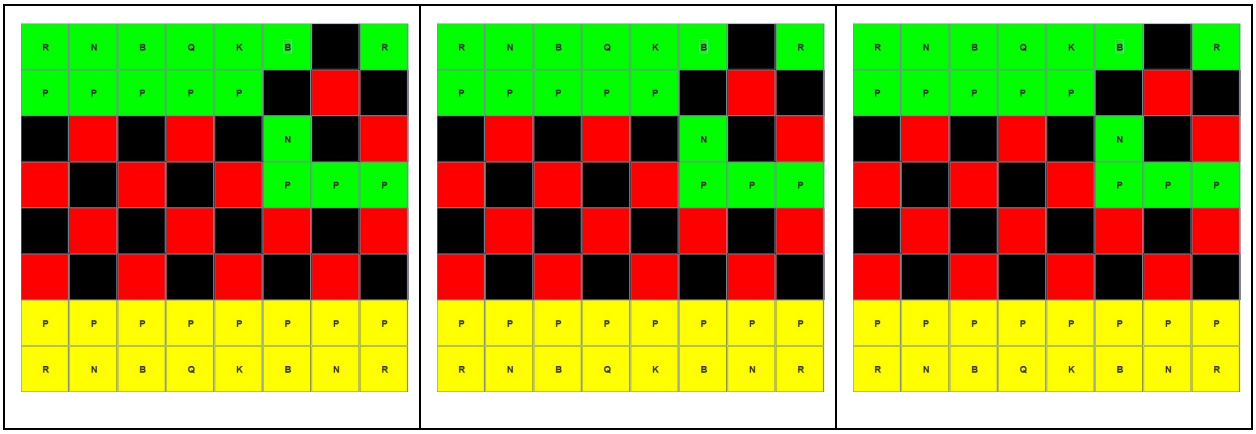
Description: We will set up the table so that this time it is not possible for the Castle to move because King has moved once.



Description: We will set up the table so that this time it is not possible for the Castle to move because Rook has moved once.

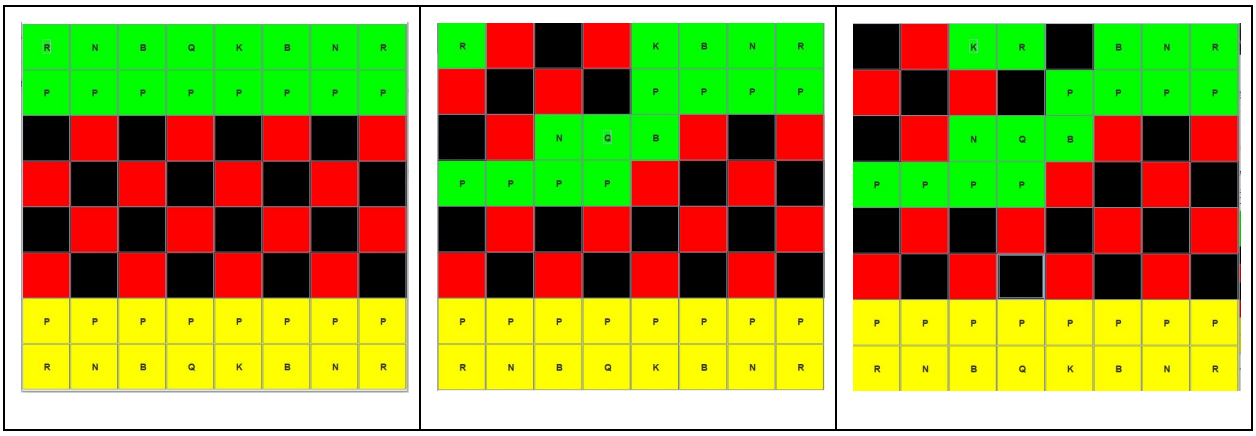


Description: We will set up the table so that this time it is not possible for the Castle to move because there is a block in the path.

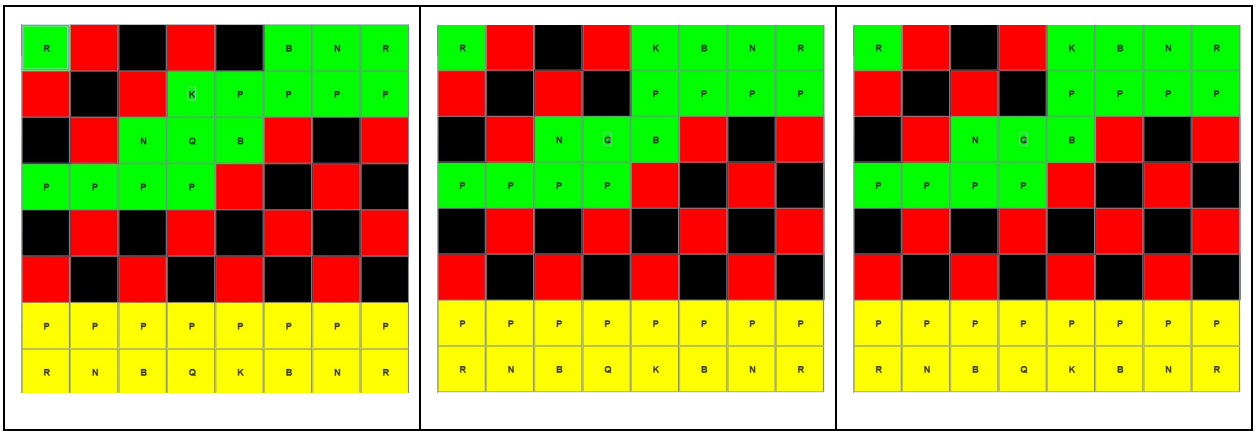


Test 5.4: North Left

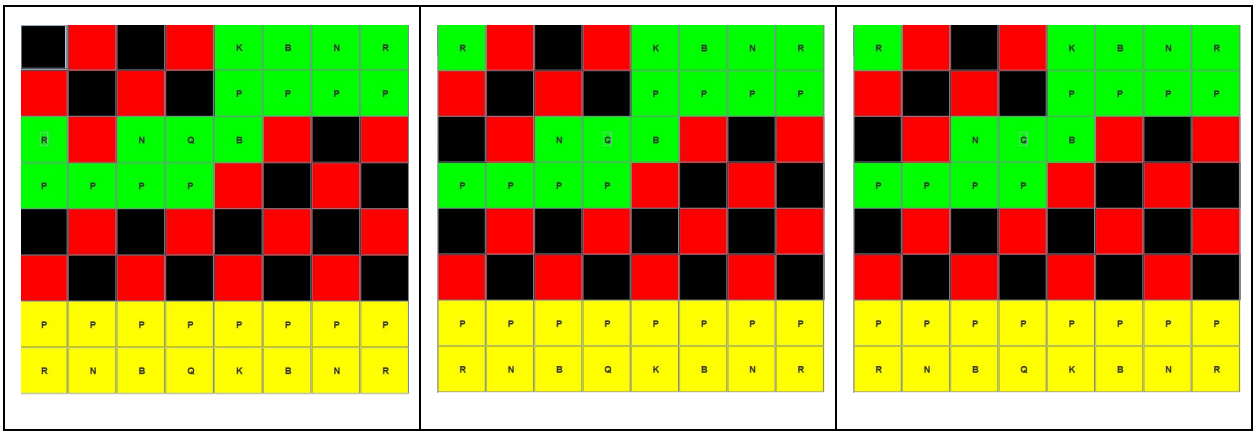
Description: We will set up the table so that this time it is possible for the Castle to move because all conditions are satisfied.



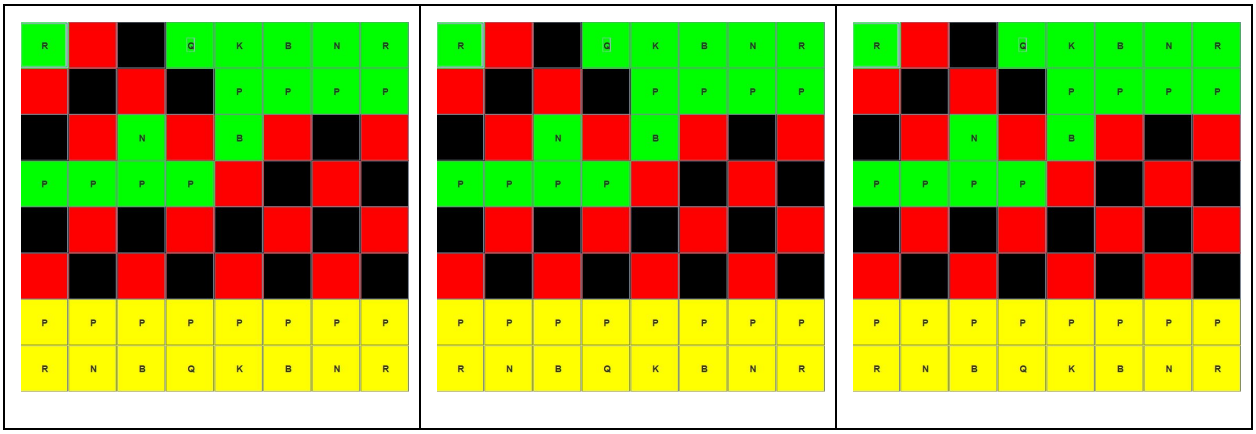
Description: We will set up the table so that this time it is not possible for the Castle to move because King has moved once.



Description: We will set up the table so that this time it is not possible for the Castle to move because Rook has moved once.

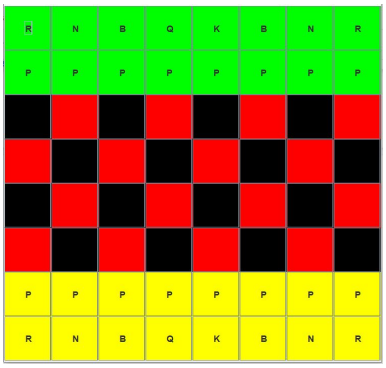


Description: We will set up the table so that this time it is not possible for the Castle to move because there is a block in the path.



6) TEST GETTER/ SETTER METHOD

Note: We will test mostly with the board fully filled, except in some cases that we need to specify how the board looks. The board will be facing north and south!



Test 6.1: testPawnSide() (PASSED)
Description: this test gets the side of Pawns on the table and compares it with the side defined by the programmer in assertEquals()

```
// compare the side of all the pawns on the table to the correct side
@Test
public void testPawnSide() {
    // initialize the complete chess board
    ChessBoard chessBoard = init( input: "CHESS");

    // compare the side of the pawn on the table to the correct side
    for (int i = 0; i < 8; ++i) {
        assertEquals(north, chessBoard.getPiece( row: 1, i).getSide());
        assertEquals(south, chessBoard.getPiece( row: 6, i).getSide());
    }
}
```

Test 6.2: testRookSide() (PASSED)
Description: this test gets the side of Rooks on the table and compares it with the side defined by the programmer in assertEquals()

```
// compare the side of all the rooks on the table to the correct side
@Test
public void testRookSide() {
    // initialize the complete chess board
    ChessBoard chessBoard = init( input: "CHESS");

    assertEquals(north, chessBoard.getPiece( row: 0,  col: 0).getSide());
    assertEquals(north, chessBoard.getPiece( row: 0,  col: 7).getSide());
    assertEquals(south, chessBoard.getPiece( row: 7,  col: 0).getSide());
    assertEquals(south, chessBoard.getPiece( row: 7,  col: 7).getSide());
}
```

Test 6.3: testKnightSide() (PASSED)
Description: this test gets the side of Knights on the table and compares it with the side defined by the programmer in assertEquals()

```
// compare the side of all the knights on the table to the correct side
@Test
public void testKnightSide() {
    // initialize the complete chess board
    ChessBoard chessBoard = init( input: "CHESS");

    assertEquals(north, chessBoard.getPiece( row: 0,  col: 1).getSide());
    assertEquals(north, chessBoard.getPiece( row: 0,  col: 6).getSide());
    assertEquals(south, chessBoard.getPiece( row: 7,  col: 1).getSide());
    assertEquals(south, chessBoard.getPiece( row: 7,  col: 6).getSide());
}
```

Test 6.4: testBishopSide() (PASSED)
Description: this test gets the side of Bishops on the table and compares it with the side defined by the programmer in assertEquals()

```
// compare the side of all the bishops on the table to the correct side
@Test
public void testBishopSide() {
    // initialize the complete chess board
    ChessBoard chessBoard = init( input: "CHESS");

    assertEquals(north, chessBoard.getPiece( row: 0,  col: 2).getSide());
    assertEquals(north, chessBoard.getPiece( row: 0,  col: 5).getSide());
    assertEquals(south, chessBoard.getPiece( row: 7,  col: 2).getSide());
    assertEquals(south, chessBoard.getPiece( row: 7,  col: 5).getSide());
}
```

Test 6.5: testQueenSide() (PASSED)
Description: this test gets the side of Queens on the table and compares it with the side defined by the programmer in assertEquals()

```
// compare the side of all the queens on the table to the correct side
@Test
public void testQueenSide() {
    // initialize the complete chess board
    ChessBoard chessBoard = init( input: "CHESS");

    assertEquals(north, chessBoard.getPiece( row: 0,  col: 3).getSide());
    assertEquals(south, chessBoard.getPiece( row: 7,  col: 3).getSide());
}
```

Test 6.6: testKingSide() (PASSED)
Description: this test gets the side of Kings on the table and compares it with the side defined by the programmer in assertEquals()

```
// compare the side of all the kings on the table to the correct side
@Test
public void testKingSide() {
    // initialize the complete chess board
    ChessBoard chessBoard = init( input: "CHESS");

    assertEquals(north, chessBoard.getPiece( row: 0,  col: 4).getSide());
    assertEquals(south, chessBoard.getPiece( row: 7,  col: 4).getSide());
}
```

Test 6.7: testPawnLabel() (PASSED)
Description: this test gets the label of Pawns on the table and compares it with the label defined by the programmer in assertEquals()

```
// compare the label of all the pawns on the table to the correct label
@Test
public void testPawnLabel() {
    // initialize the complete chess board
    ChessBoard chessBoard = init( input: "CHESS");

    // compare the label of the pawn on the table to the correct label
    for (int i = 0; i < 8; ++i) {
        assertEquals( expected: "P", chessBoard.getPiece( row: 1, i).getLabel());
        assertEquals( expected: "P", chessBoard.getPiece( row: 6, i).getLabel());
    }
}
```

Test 6.8: testRookLabel() (PASSED)
Description: this test gets the label of Rooks on the table and compares it with the label defined by the programmer in assertEquals()

```
// compare the label of all the rooks on the table to the correct label
@Test
public void testRookLabel() {
    // initialize the complete chess board
    ChessBoard chessBoard = init( input: "CHESS");

    assertEquals( expected: "R", chessBoard.getPiece( row: 0,  col: 0).getLabel());
    assertEquals( expected: "R", chessBoard.getPiece( row: 0,  col: 7).getLabel());
    assertEquals( expected: "R", chessBoard.getPiece( row: 7,  col: 0).getLabel());
    assertEquals( expected: "R", chessBoard.getPiece( row: 7,  col: 7).getLabel());
}
```

Test 6.9: testKnightLabel() (PASSED)
Description: this test gets the label of Knights on the table and compares it with the label defined by the programmer in assertEquals()

```
// compare the label of all the knights on the table to the correct label
@Test
public void testKnightLabel() {
    // initialize the complete chess board
    ChessBoard chessBoard = init( input: "CHESS");

    assertEquals( expected: "N", chessBoard.getPiece( row: 0,  col: 1).getLabel());
    assertEquals( expected: "N", chessBoard.getPiece( row: 0,  col: 6).getLabel());
    assertEquals( expected: "N", chessBoard.getPiece( row: 7,  col: 1).getLabel());
    assertEquals( expected: "N", chessBoard.getPiece( row: 7,  col: 6).getLabel());
}
```

Test 6.10: testBishopLabel() (PASSED)
Description: this test gets the label of Bishops on the table and compares it with the label defined by the programmer in assertEquals()

```
// compare the label of all the bishops on the table to the correct label
@Test
public void testBishopLabel() {
    // initialize the complete chess board
    ChessBoard chessBoard = init( input: "CHESS");

    assertEquals( expected: "B", chessBoard.getPiece( row: 0,  col: 2).getLabel());
    assertEquals( expected: "B", chessBoard.getPiece( row: 0,  col: 5).getLabel());
    assertEquals( expected: "B", chessBoard.getPiece( row: 7,  col: 2).getLabel());
    assertEquals( expected: "B", chessBoard.getPiece( row: 7,  col: 5).getLabel());
}
```

Test 6.11: testQueenLabel() (PASSED)
Description: this test gets the label of Queens on the table and compares it with the label defined by the programmer in assertEquals()

```
// compare the label of all the queens on the table to the correct label
@Test
public void testQueenLabel() {
    // initialize the complete chess board
    ChessBoard chessBoard = init( input: "CHESS");

    assertEquals( expected: "Q", chessBoard.getPiece( row: 0,  col: 3).getLabel());
    assertEquals( expected: "Q", chessBoard.getPiece( row: 7,  col: 3).getLabel());
}
```

Test 6.12: testKingLabel() (PASSED)
Description: this test gets the label of Kings on the table and compares it with the label defined by the programmer in assertEquals()

```
// compare the label of all the kings on the table to the correct label
@Test
public void testKingLabel() {
    // initialize the complete chess board
    ChessBoard chessBoard = init( input: "CHESS");

    assertEquals( expected: "K", chessBoard.getPiece( row: 0,  col: 4).getLabel());
    assertEquals( expected: "K", chessBoard.getPiece( row: 7,  col: 4).getLabel());
}
```

Test 6.13: testNullIcon() (PASSED)
Description: this test gets the icon of pieces on the table and compares it with null.

```
// compare the icons of all the pieces on the table to the correct icons
@Test
public void testNullIcon() {
    // initialize the complete chess board
    ChessBoard chessBoard = init( input: "CHESS");

    // compare the icon of the piece on the table to the correct icon
    for (int i = 0; i < 8; ++i) {
        assertEquals( expected: null, chessBoard.getPiece( row: 0, i).getIcon());
        assertEquals( expected: null, chessBoard.getPiece( row: 1, i).getIcon());
        assertEquals( expected: null, chessBoard.getPiece( row: 6, i).getIcon());
        assertEquals( expected: null, chessBoard.getPiece( row: 7, i).getIcon());
    }
}
```

Test 6.14: testPawnPosition() (PASSED)
Description: this test gets the row and column of Pawns on the table and compares it with the row and column defined by the programmer in assertEquals()

```
// compare the positions of all the pawns on the table to the correct positions
@Test
public void testPawnPosition() {
    // initialize the complete chess board
    ChessBoard chessBoard = init( input: "CHESS");

    // compare the position of the pawn on the table to the correct position
    for (int i = 0; i < 8; ++i) {
        assertEquals( expected: 1, chessBoard.getPiece( row: 1, i).getRow());
        assertEquals(i, chessBoard.getPiece( row: 1, i).getColumn());

        assertEquals( expected: 6, chessBoard.getPiece( row: 6, i).getRow());
        assertEquals(i, chessBoard.getPiece( row: 6, i).getColumn());
    }
}
```

Test 6.15: testRookPosition() (PASSED)
Description: this test gets the row and column of Rooks on the table and compares it with the row and column defined by the programmer in assertEquals()

```
// compare the positions of all the rooks on the table to the correct positions
@Test
public void testRookPosition() {
    // initialize the complete chess board
    ChessBoard chessBoard = init( input: "CHESS");

    assertEquals( expected: 0, chessBoard.getPiece( row: 0,  col: 0).getRow());
    assertEquals( expected: 0, chessBoard.getPiece( row: 0,  col: 0).getColumn());

    assertEquals( expected: 0, chessBoard.getPiece( row: 0,  col: 7).getRow());
    assertEquals( expected: 7, chessBoard.getPiece( row: 0,  col: 7).getColumn());

    assertEquals( expected: 7, chessBoard.getPiece( row: 7,  col: 0).getRow());
    assertEquals( expected: 0, chessBoard.getPiece( row: 7,  col: 0).getColumn());

    assertEquals( expected: 7, chessBoard.getPiece( row: 7,  col: 7).getRow());
    assertEquals( expected: 7, chessBoard.getPiece( row: 7,  col: 7).getColumn());
}
```

Test 6.16: testKnightPosition() (PASSED)
Description: this test gets the row and column of Knights on the table and compares it with the row and column defined by the programmer in assertEquals()

```
// compare the positions of all the knights on the table to the correct positions
@Test
public void testKnightPosition() {
    // initialize the complete chess board
    ChessBoard chessBoard = init( input: "CHESS");

    assertEquals( expected: 0, chessBoard.getPiece( row: 0,  col: 1).getRow());
    assertEquals( expected: 1, chessBoard.getPiece( row: 0,  col: 1).getColumn());

    assertEquals( expected: 0, chessBoard.getPiece( row: 0,  col: 6).getRow());
    assertEquals( expected: 6, chessBoard.getPiece( row: 0,  col: 6).getColumn());

    assertEquals( expected: 7, chessBoard.getPiece( row: 7,  col: 1).getRow());
    assertEquals( expected: 1, chessBoard.getPiece( row: 7,  col: 1).getColumn());

    assertEquals( expected: 7, chessBoard.getPiece( row: 7,  col: 6).getRow());
    assertEquals( expected: 6, chessBoard.getPiece( row: 7,  col: 6).getColumn());
}
```

Test 6.17: testBishopPosition() (PASSED)
Description: this test gets the row and column of Bishops on the table and compares it with the row and column defined by the programmer in assertEquals()

```
// compare the positions of all the bishops on the table to the correct positions
@Test
public void testBishopPosition() {
    // initialize the complete chess board
    ChessBoard chessBoard = init( input: "CHESS");

    assertEquals( expected: 0, chessBoard.getPiece( row: 0,  col: 2).getRow());
    assertEquals( expected: 2, chessBoard.getPiece( row: 0,  col: 2).getColumn());

    assertEquals( expected: 0, chessBoard.getPiece( row: 0,  col: 5).getRow());
    assertEquals( expected: 5, chessBoard.getPiece( row: 0,  col: 5).getColumn());

    assertEquals( expected: 7, chessBoard.getPiece( row: 7,  col: 2).getRow());
    assertEquals( expected: 2, chessBoard.getPiece( row: 7,  col: 2).getColumn());

    assertEquals( expected: 7, chessBoard.getPiece( row: 7,  col: 5).getRow());
    assertEquals( expected: 5, chessBoard.getPiece( row: 7,  col: 5).getColumn());
}
```

Test 6.18: testQueenPosition() (PASSED)
Description: this test gets the row and column of Queens on the table and compares it with the row and column defined by the programmer in assertEquals()

```
// compare the positions of all the queens on the table to the correct positions
@Test
public void testQueenPosition() {
    // initialize the complete chess board
    ChessBoard chessBoard = init( input: "CHESS");

    assertEquals( expected: 0, chessBoard.getPiece( row: 0,  col: 3).getRow());
    assertEquals( expected: 3, chessBoard.getPiece( row: 0,  col: 3).getColumn());

    assertEquals( expected: 7, chessBoard.getPiece( row: 7,  col: 3).getRow());
    assertEquals( expected: 3, chessBoard.getPiece( row: 7,  col: 3).getColumn());
}
```

Test 6.19: testKingPosition() (PASSED)
Description: this test gets the row and column of Queens on the table and compares it with the row and column defined by the programmer in assertEquals()

```
// compare the positions of all the kings on the table to the correct positions
@Test
public void testKingPosition() {
    // initialize the complete chess board
    ChessBoard chessBoard = init( input: "CHESS");

    assertEquals( expected: 0, chessBoard.getPiece( row: 0,  col: 4).getRow());
    assertEquals( expected: 4, chessBoard.getPiece( row: 0,  col: 4).getColumn());

    assertEquals( expected: 7, chessBoard.getPiece( row: 7,  col: 4).getRow());
    assertEquals( expected: 4, chessBoard.getPiece( row: 7,  col: 4).getColumn());
}
```

Test 6.20: testChessPieceConstructor() (PASSED)
Description: this test creates a new piece for all kinds of pieces and compare it with its board as well as label

- Test pawn constructor:

```
// test pawn constructor
PawnPiece pawn = new PawnPiece(north, chessBoard);
assertEquals(north, pawn.getSide());
assertEquals( expected: "P", pawn.getLabel());
```

- Test rook constructor

```
// test rook constructor
RookPiece rook = new RookPiece(south, chessBoard);
assertEquals(south, rook.getSide());
assertEquals( expected: "R", rook.getLabel());
```

- Test king constructor

```
// test king constructor
KingPiece king = new KingPiece(east, chessBoard);
assertEquals(east, king.getSide());
assertEquals( expected: "K", king.getLabel());
```

- Test queen constructor

```
// test queen constructor
QueenPiece queen = new QueenPiece(west, chessBoard);
assertEquals(west, queen.getSide());
assertEquals( expected: "Q", queen.getLabel());
```

- Test knight constructor

```
// test knight constructor
KnightPiece knight = new KnightPiece(north, chessBoard);
assertEquals(north, knight.getSide());
assertEquals( expected: "N", knight.getLabel());
```

- Test bishop constructor

```
// test bishop constructor
BishopPiece bishop = new BishopPiece(south, chessBoard);
assertEquals(south, bishop.getSide());
assertEquals( expected: "B", bishop.getLabel());
```

Test 6.20: testBoard() (PASSED)
Description: this test creates a new board for all kinds of board sides and compares with the correct side defined

- Test north board:

```
// test north board
PawnPiece northBoard = new PawnPiece(north, chessBoard);
assertEquals(north, northBoard.getSide());
```

- Test south board:

```
// test south board
PawnPiece southBoard = new PawnPiece(south, chessBoard);
assertEquals(south, southBoard.getSide());
```

- Test east board:

```
// test east board
PawnPiece eastBoard = new PawnPiece(east, chessBoard);
assertEquals(east, eastBoard.getSide());
```

- Test north board:

```
// test west board
PawnPiece westBoard = new PawnPiece(west, chessBoard);
assertEquals(west, westBoard.getSide());
```

Test 6.20: testFirstPlayer() (PASSED)
Description: I created a new method in the ChessGame interface to get the first player. This test will return true if the first player's side matches the programmer's input.

- Test north being the first player:

```
// test north being the first player
ChessBoard northBoard = init( input: "CHESS", north);
assertEquals(north, northBoard.getGameRules().getFirstPlayer());
```

- Test south being the first player:

```
// test south being the first player
ChessBoard southBoard = init( input: "CHESS", south);
assertEquals(south, southBoard.getGameRules().getFirstPlayer());
```

- Test east being the first player:

```
// test east being the first player
ChessBoard eastBoard = init( input: "CHESS", east);
assertEquals(east, eastBoard.getGameRules().getFirstPlayer());
```

- Test west being the first player:

```
// test west being the first player
ChessBoard westBoard = init( input: "CHESS", west);
assertEquals(west, westBoard.getGameRules().getFirstPlayer());
```

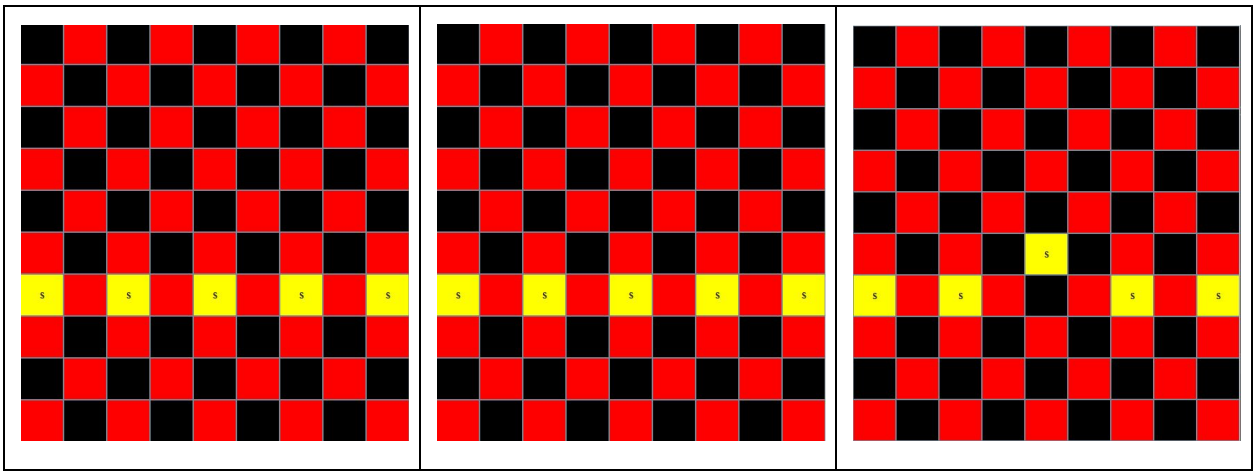
XIANQI: most of the pieces were tested thoroughly above, we will test how the pieces move and how it functions.

7) TEST XIANQI CHESS PIECE MOVEMENT

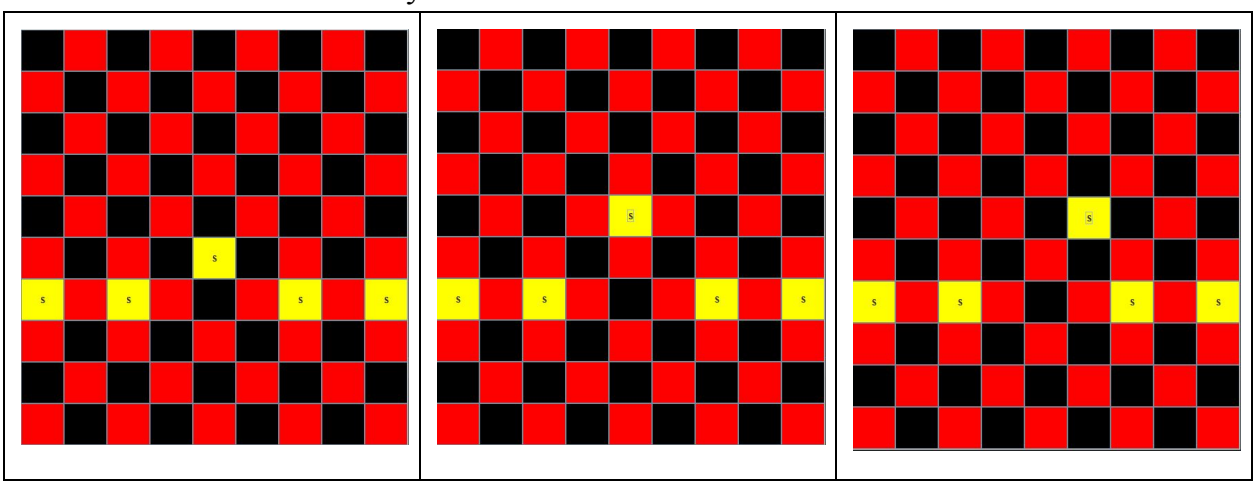
Note: we will consider every row to be the line, and the river is omitted.

Test 7.1: test SoldierPiece move (PASSED)

Description: If we try to move the Soldier horizontally without passing the boundary or backward to its side, the Soldier will stand still. After the Soldier at position (6, 4) moves. It will reach row 5.

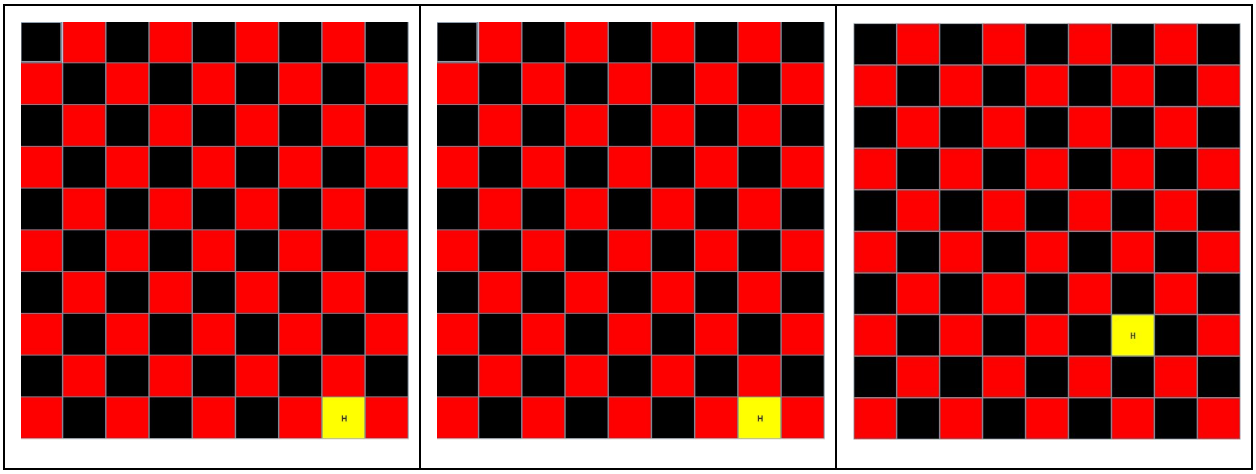


Description: We will make another move to cross the river. When this happens, we can move the soldiers horizontally.

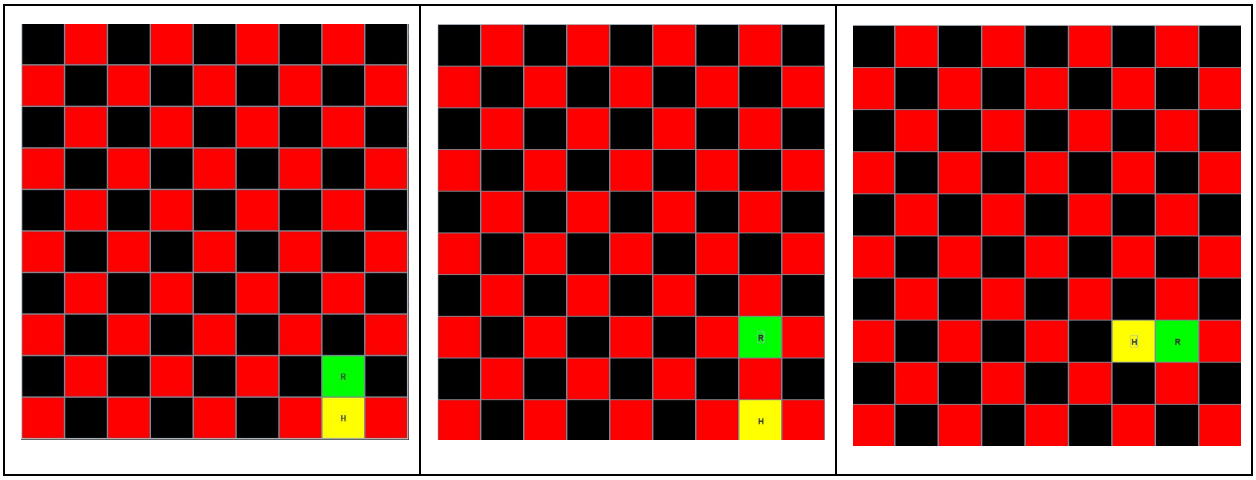


Test 7.2: test HorsePiece move (PASSED)

Description: When we try to move somewhere not in the L-path, the Horse will stand at its own place. However, when we move the Knight from (9, 7) to (7,6), it is a L-path, which is valid.

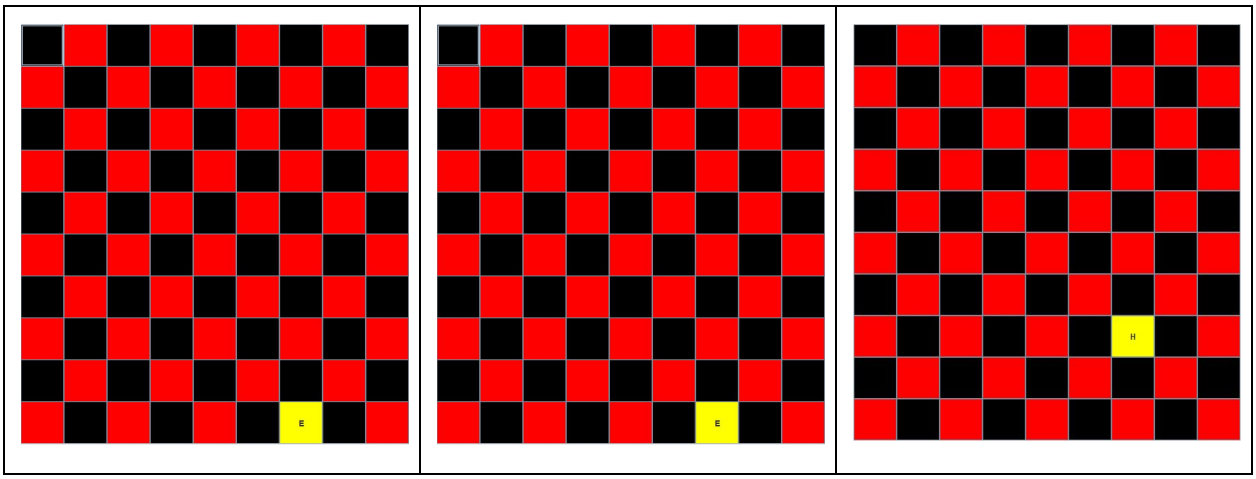


Description: When we try to move the Horse in a L-path, it will stand still because it is blocked by the Rook. Nonetheless, if the Rook is also on a L-path but there is an empty space between the Horse and the Rook, the Horse can move.

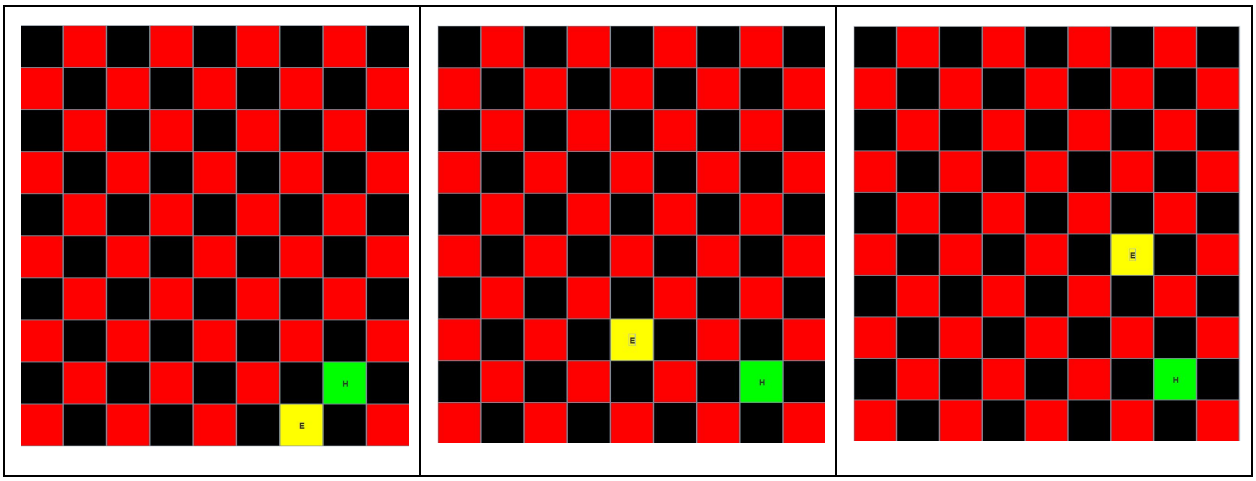


Test 7.3: test ElephantPiece move (PASSED)

Description: When we try to move somewhere not in the 2-step diagonal path, the Elephant will stand at its own place. We will make a valid move from (9,6) to (7,8).



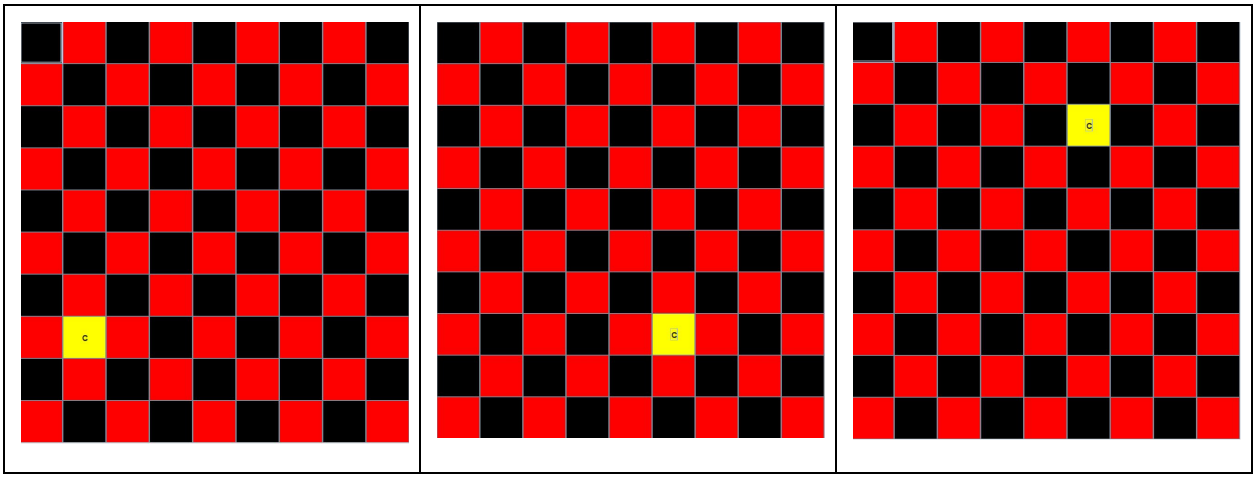
Description: When we try to move while the Elephant is blocked, it will stand still. Also, when we try to cross the river at point (5,6), it will stand still.



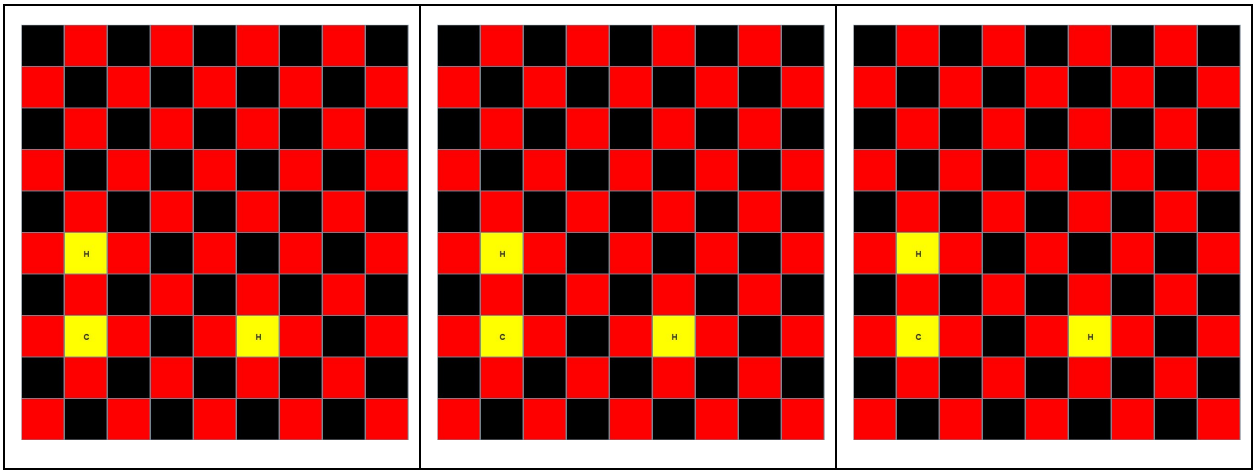
Test 7.4: test RookPiece move (PASSED) - similar to European Chess's Rook.

Test 7.5: test CannonPiece move (PASSED)

Description: When we try to move somewhere not in the horizontal or vertical path, the Cannon will stand at its own place. We will make 2 valid moves: horizontal from (7,1) to (7,5), and vertical from (7,5) to (2,5).

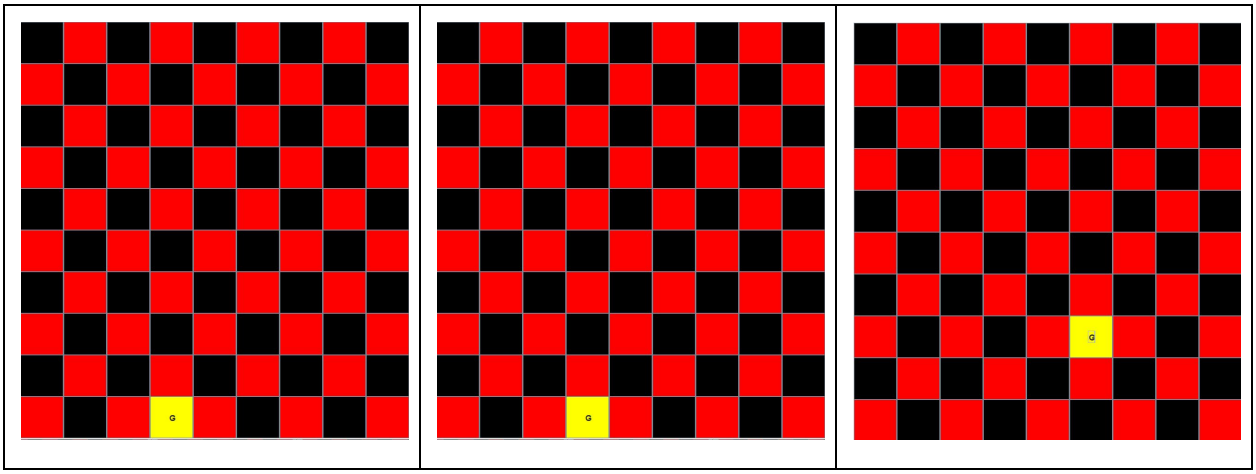


Description: When we try to move while the Cannon is blocked, it will stand still. When we try to move from (7,1) to (7, 6) or (7,1) to (4,1), it is an invalid path. We will test capture later on.

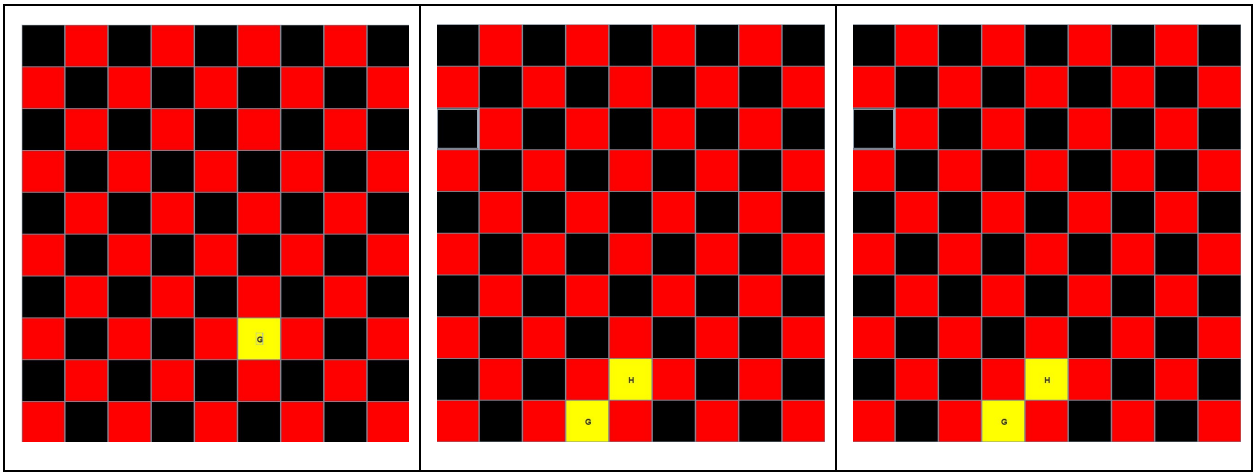


Test 7.6: test GuardPiece move (PASSED)

Description: When we try to move somewhere not in the diagonal path, the Guard will stand at its own place. We will move 2 moves diagonally to move the Guard from (9, 3) to (8, 4) to (7, 5).

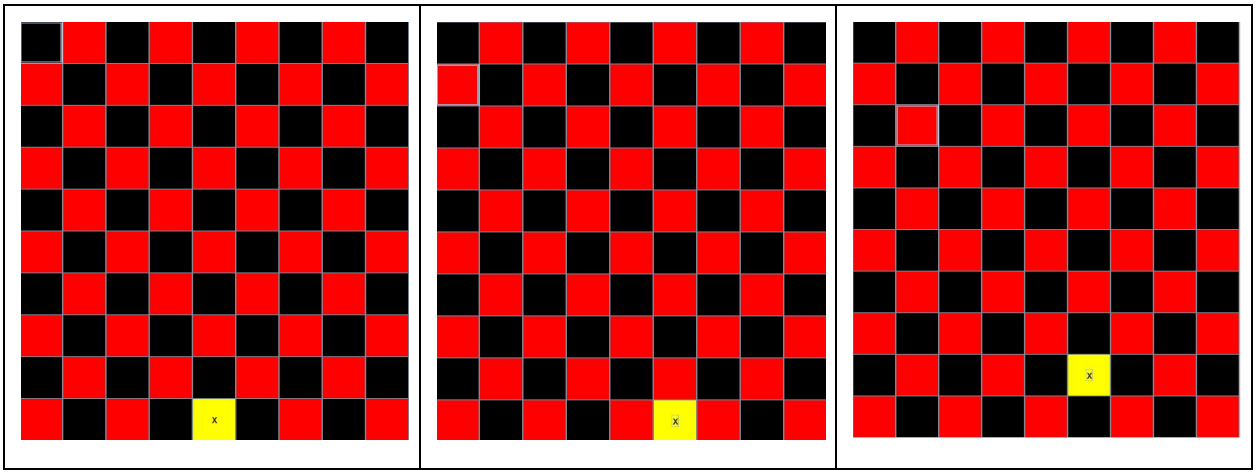


Description: When we try to move outside the palace, the Guard will stand still. Also when we try to move the guard while it's blocked diagonally, it will also stand still.

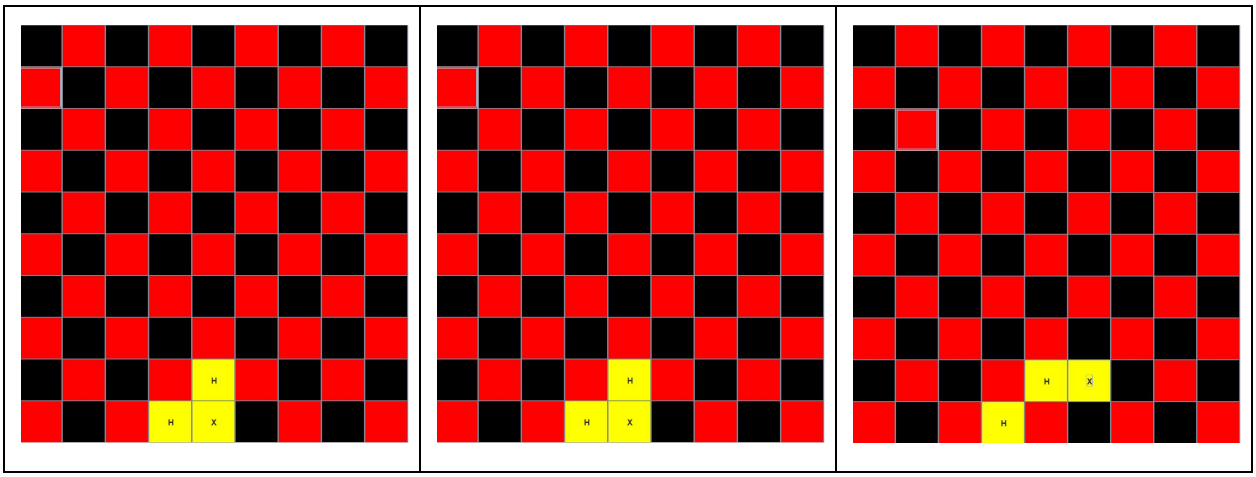


Test 7.7: test XianqiKingPiece move (PASSED)

Description: When we try to move somewhere over 2 squares not in the diagonal path, the King will stand at its own place. We will move 3 moves to move the King: horizontally from (9, 4) to (9, 5), vertically from (9, 5) to (8, 5), and diagonally from (8, 5) to (9, 4) (in this step, it will return to the original position).



Description: When we try to move outside the palace, the King will stand still. In this case, the King is blocked horizontally and vertically, but not diagonally. Thus we can move the King from (9, 4) to (8, 5).

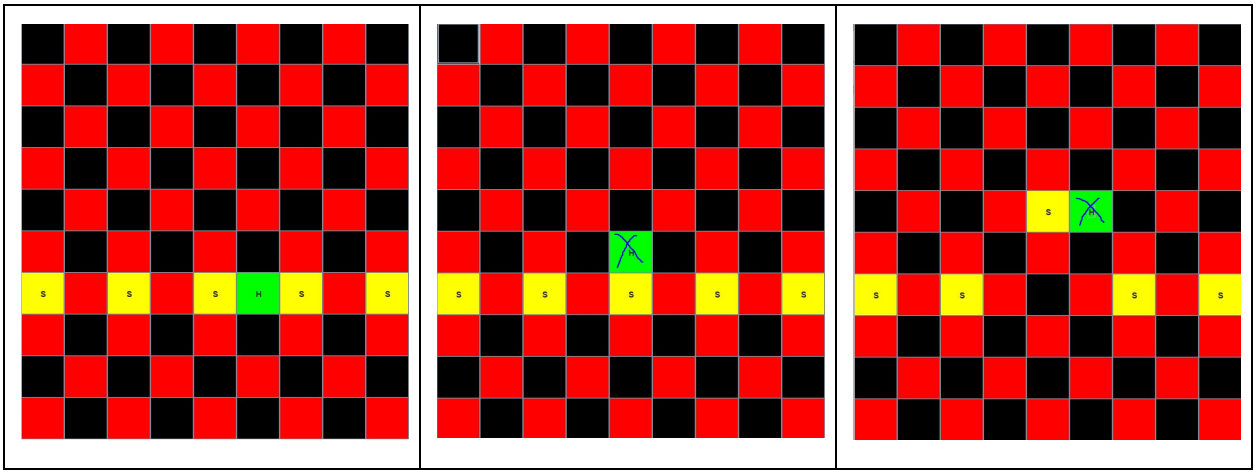


8) TEST XIANQI CHESS PIECE CAPTURE

Test 8.1: test SoldierPiece capture (PASSED)

Description: We will test the capture move when it has not crossed the river and when it does, which makes the capture legal horizontally.

- In the first picture, it is impossible for the soldier to capture the horse because the soldier has not crossed the river.
- In the second picture, however, the soldier can because it is a vertical capture.
- In the third picture, since the soldier has crossed the river, it can capture the horse horizontally



Test 8.2: test HorsePiece capture (PASSED) - this will function the same as the Knight tested above, but this time, it can only capture if not blocked.

Test 8.3: test ElephantPiece capture (PASSED) - this will function the same as the Bishop tested above, but this time, it can only capture in diagonal direction and must be 2 squares.

Test 8.4: test RookPiece capture (PASSED) - this will function the same as the Rook tested above.

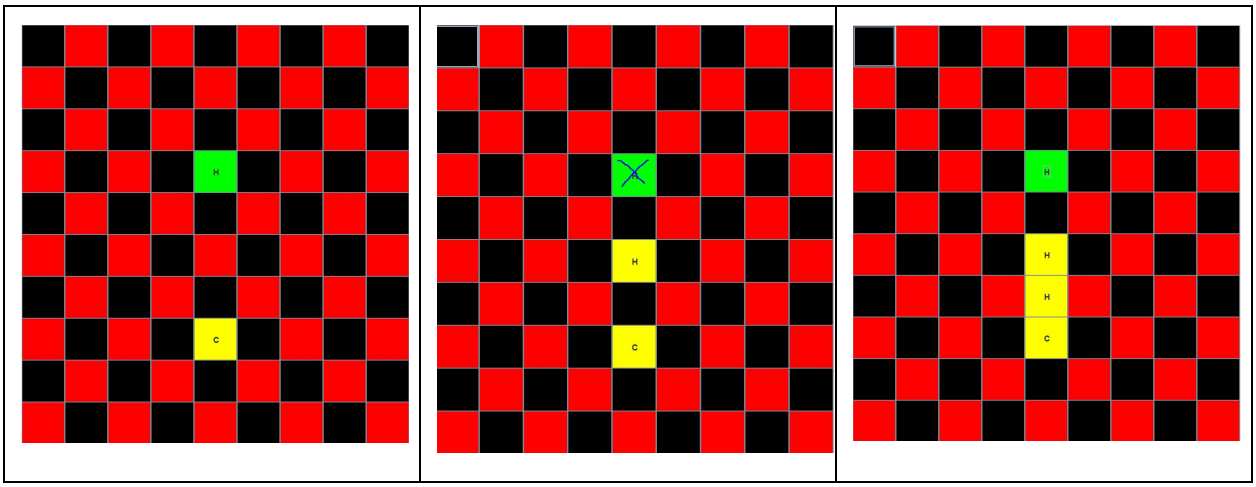
Test 8.5: test GuardPiece capture (PASSED) - this will function the same as the Bishop tested above, but this time, it can only capture while in the palace and in diagonal direction.

Test 8.6: test XianqiKingPiece capture (PASSED) - this will function the same as the King tested above, but this time, it can only capture while in the palace.

Test 8.7: test CannonPiece capture (PASSED)

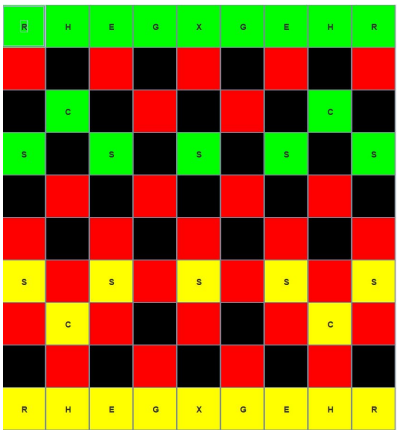
Description: We will test the capture move when it is not blocked on its path and the piece to be captured must be of different side. And then we test when it is blocked by one piece, following with a test blocked by more than 1 piece.

- In the first picture, it is impossible for the cannon to capture the horse because it is not blocked.
- In the second picture, however, the cannon can because it is a vertical capture.
- In the third picture, it is impossible for the cannon to capture the horse because it is blocked by more than one piece.



9) TEST GETTER/ SETTER METHOD

Note: We will test mostly with the board fully filled, except in some cases that we need to specify how the board looks. The board will be facing north and south!



Test 9.1: testSoldierSide() (PASSED)
Description: this test gets the side of Soldiers on the table and compares it with the side defined by the programmer in assertEquals()

```
// compare the side of all the pawns on the table to the correct side
@Test
public void testSoldierSide() {
    // initialize the complete chess board
    ChessBoard chessBoard = init( input: "XIANQI");

    // compare the side of the soldier on the table to the correct side
    for (int i = 0; i < 9; ++i) {
        if (i % 2 == 0) {
            assertEquals(north, chessBoard.getPiece( row: 3, i).getSide());
            assertEquals(south, chessBoard.getPiece( row: 6, i).getSide());
        }
    }
}
```

Test 9.2: testHorseSide() (PASSED)
Description: this test gets the side of Horses on the table and compares it with the side defined by the programmer in assertEquals()

```
// compare the side of all the horses on the table to the correct side
@Test
public void testHorseSide() {
    // initialize the complete chess board
    ChessBoard chessBoard = init( input: "XIANQI");

    assertEquals(north, chessBoard.getPiece( row: 0,  col: 1).getSide());
    assertEquals(north, chessBoard.getPiece( row: 0,  col: 7).getSide());
    assertEquals(south, chessBoard.getPiece( row: 9,  col: 1).getSide());
    assertEquals(south, chessBoard.getPiece( row: 9,  col: 7).getSide());
}
```

Test 9.3: testElephantSide() (PASSED)
Description: this test gets the side of Elephants on the table and compares it with the side defined by the programmer in assertEquals()

```
// compare the side of all the elephants on the table to the correct side
@Test
public void testElephantSide() {
    // initialize the complete chess board
    ChessBoard chessBoard = init( input: "XIANQI");

    assertEquals(north, chessBoard.getPiece( row: 0,  col: 2).getSide());
    assertEquals(north, chessBoard.getPiece( row: 0,  col: 6).getSide());
    assertEquals(south, chessBoard.getPiece( row: 9,  col: 2).getSide());
    assertEquals(south, chessBoard.getPiece( row: 9,  col: 6).getSide());
}
```

Test 9.4: testRookSide() (PASSED)
Description: this test gets the side of Rooks on the table and compares it with the side defined by the programmer in assertEquals()

```
// compare the side of all the rooks on the table to the correct side
@Test
public void testRookSide() {
    // initialize the complete chess board
    ChessBoard chessBoard = init( input: "XIANQI");

    assertEquals(north, chessBoard.getPiece( row: 0,  col: 0).getSide());
    assertEquals(north, chessBoard.getPiece( row: 0,  col: 8).getSide());
    assertEquals(south, chessBoard.getPiece( row: 9,  col: 0).getSide());
    assertEquals(south, chessBoard.getPiece( row: 9,  col: 8).getSide());
}
```

Test 9.5: testCannonSide() (PASSED)
Description: this test gets the side of Kings on the table and compares it with the side defined by the programmer in assertEquals()

```
// compare the side of all the cannons on the table to the correct side
@Test
public void testCannonSide() {
    // initialize the complete chess board
    ChessBoard chessBoard = init( input: "XIANQI");

    assertEquals(north, chessBoard.getPiece( row: 2,  col: 1).getSide());
    assertEquals(north, chessBoard.getPiece( row: 2,  col: 7).getSide());
    assertEquals(south, chessBoard.getPiece( row: 7,  col: 1).getSide());
    assertEquals(south, chessBoard.getPiece( row: 7,  col: 7).getSide());
}
```

Test 9.6: testGuardSide() (PASSED)
Description: this test gets the side of Guards on the table and compares it with the side defined by the programmer in assertEquals()

```
// compare the side of all the bishops on the table to the correct side
@Test
public void testBishopSide() {
    // initialize the complete chess board
    ChessBoard chessBoard = init( input: "CHESS");

    assertEquals(north, chessBoard.getPiece( row: 0,  col: 2).getSide());
    assertEquals(north, chessBoard.getPiece( row: 0,  col: 5).getSide());
    assertEquals(south, chessBoard.getPiece( row: 7,  col: 2).getSide());
    assertEquals(south, chessBoard.getPiece( row: 7,  col: 5).getSide());
}
```

Test 9.7: testXianqiKingSide() (PASSED)
Description: this test gets the side of Kings on the table and compares it with the side defined by the programmer in assertEquals()

```
// compare the side of all the xiangi kings on the table to the correct side
@Test
public void testXianqiKingSide() {
    // initialize the complete chess board
    ChessBoard chessBoard = init( input: "XIANQI");

    assertEquals(north, chessBoard.getPiece( row: 0,  col: 4).getSide());
    assertEquals(south, chessBoard.getPiece( row: 9,  col: 4).getSide());
}
```

Test 9.8: testSoldierLabel() (PASSED)
Description: this test gets the label of Soldiers on the table and compares it with the label defined by the programmer in assertEquals()

```
// compare the label of all the soldiers on the table to the correct label
@Test
public void testSoldierLabel() {
    // initialize the complete chess board
    ChessBoard chessBoard = init( input: "XIANQI");

    // compare the side of the soldier on the table to the correct label
    for (int i = 0; i < 9; ++i) {
        if (i % 2 == 0) {
            assertEquals( expected: "S", chessBoard.getPiece( row: 3, i).getLabel());
            assertEquals( expected: "S", chessBoard.getPiece( row: 6, i).getLabel());
        }
    }
}
```

Test 9.9: testHorseLabel() (PASSED)
Description: this test gets the label of Horses on the table and compares it with the label defined by the programmer in assertEquals()

```
// compare the side of all the horses on the table to the correct label
@Test
public void testHorseLabel() {
    // initialize the complete chess board
    ChessBoard chessBoard = init( input: "XIANQI");

    assertEquals( expected: "H", chessBoard.getPiece( row: 0,  col: 1).getLabel());
    assertEquals( expected: "H", chessBoard.getPiece( row: 0,  col: 7).getLabel());
    assertEquals( expected: "H", chessBoard.getPiece( row: 9,  col: 1).getLabel());
    assertEquals( expected: "H", chessBoard.getPiece( row: 9,  col: 7).getLabel());
}
```

Test 9.10: testElephantLabel() (PASSED)
Description: this test gets the label of Elephants on the table and compares it with the label defined by the programmer in assertEquals()

```
// compare the side of all the elephants on the table to the correct label
@Test
public void testElephantLabel() {
    // initialize the complete chess board
    ChessBoard chessBoard = init( input: "XIANQI");

    assertEquals( expected: "E", chessBoard.getPiece( row: 0,  col: 2).getLabel());
    assertEquals( expected: "E", chessBoard.getPiece( row: 0,  col: 6).getLabel());
    assertEquals( expected: "E", chessBoard.getPiece( row: 9,  col: 2).getLabel());
    assertEquals( expected: "E", chessBoard.getPiece( row: 9,  col: 6).getLabel());
}
```

Test 9.11: testRookLabel() (PASSED)
Description: this test gets the label of Rooks on the table and compares it with the label defined by the programmer in assertEquals()

```
// compare the label of all the rooks on the table to the correct label
@Test
public void testRookLabel() {
    // initialize the complete chess board
    ChessBoard chessBoard = init( input: "XIANQI");

    assertEquals( expected: "R", chessBoard.getPiece( row: 0,  col: 0).getLabel());
    assertEquals( expected: "R", chessBoard.getPiece( row: 0,  col: 8).getLabel());
    assertEquals( expected: "R", chessBoard.getPiece( row: 9,  col: 0).getLabel());
    assertEquals( expected: "R", chessBoard.getPiece( row: 9,  col: 8).getLabel());
}
```

Test 9.12: testCannonLabel() (PASSED)
Description: this test gets the label of Cannons on the table and compares it with the label defined by the programmer in assertEquals()

```
// compare the side of all the cannons on the table to the correct label
@Test
public void testCannonLabel() {
    // initialize the complete chess board
    ChessBoard chessBoard = init( input: "XIANQI");

    assertEquals( expected: "C", chessBoard.getPiece( row: 2,  col: 1).getLabel());
    assertEquals( expected: "C", chessBoard.getPiece( row: 2,  col: 7).getLabel());
    assertEquals( expected: "C", chessBoard.getPiece( row: 7,  col: 1).getLabel());
    assertEquals( expected: "C", chessBoard.getPiece( row: 7,  col: 7).getLabel());
}
```

Test 9.13: testGuardLabel() (PASSED)
Description: this test gets the label of Guards on the table and compares it with the label defined by the programmer in assertEquals()

```
// compare the label of all the guards on the table to the correct label
@Test
public void testGuardLabel() {
    // initialize the complete chess board
    ChessBoard chessBoard = init( input: "XIANQI");

    assertEquals( expected: "G", chessBoard.getPiece( row: 0,  col: 3).getLabel());
    assertEquals( expected: "G", chessBoard.getPiece( row: 0,  col: 5).getLabel());
    assertEquals( expected: "G", chessBoard.getPiece( row: 9,  col: 3).getLabel());
    assertEquals( expected: "G", chessBoard.getPiece( row: 9,  col: 5).getLabel());
}
```

Test 9.14: testXianqiKingLabel() (PASSED)
Description: this test gets the label of Kings on the table and compares it with the label defined by the programmer in assertEquals()

```
// compare the label of all the xiangi kings on the table to the correct label
@Test
public void testXiangiKingLabel() {
    // initialize the complete chess board
    ChessBoard chessBoard = init( input: "XIANQI");

    assertEquals( expected: "X", chessBoard.getPiece( row: 0,  col: 4).getLabel());
    assertEquals( expected: "X", chessBoard.getPiece( row: 9,  col: 4).getLabel());
}
```

Test 9.15: testNullIcon() (PASSED)
Description: this test gets the icon of pieces on the table and compares it with null.

```
// compare the icons of all the pieces on the table to the correct icons
@Test
public void testNullIcon() {
    // initialize the complete chess board
    ChessBoard chessBoard = init( input: "XIANQI");

    // compare the icon of the piece on the table to the correct icon
    for (int i = 0; i < 9; ++i) {
        assertEquals( expected: null, chessBoard.getPiece( row: 0, i).getIcon());
        assertEquals( expected: null, chessBoard.getPiece( row: 9, i).getIcon());
        if (i % 2 == 0) {
            assertEquals( expected: null, chessBoard.getPiece( row: 3, i).getIcon());
            assertEquals( expected: null, chessBoard.getPiece( row: 6, i).getIcon());
        }
    }
}
```

Test 9.16: testSoldierPosition() (PASSED)
Description: this test gets the row and column of Soldiers on the table and compares it with the row and column defined by the programmer in assertEquals()

```
// compare the positions of all the soldiers on the table to the correct positions
@Test
public void testSoldierPosition() {
    // initialize the complete chess board
    ChessBoard chessBoard = init( input: "XIANQI");

    // compare the position of the soldier on the table to the correct position
    for (int i = 0; i < 9; ++i) {
        if (i % 2 == 0) {
            ChessPiece northPiece = chessBoard.getPiece( row: 3, i);
            assertEquals( expected: 3, northPiece.getRow());
            assertEquals(i, northPiece.getColumn());

            ChessPiece southPiece = chessBoard.getPiece( row: 6, i);
            assertEquals( expected: 6, southPiece.getRow());
            assertEquals(i, southPiece.getColumn());
        }
    }
}
```

Test 9.17: testHorsePosition() (PASSED)
Description: this test gets the row and column of Horses on the table and compares it with the row and column defined by the programmer in assertEquals()

```
// compare the side of all the horses on the table to the correct positions
@Test
public void testHorsePosition() {
    // initialize the complete chess board
    ChessBoard chessBoard = init( input: "XIANQI");

    assertEquals( expected: 0, chessBoard.getPiece( row: 0,  col: 1).getRow());
    assertEquals( expected: 1, chessBoard.getPiece( row: 0,  col: 1).getColumn());

    assertEquals( expected: 0, chessBoard.getPiece( row: 0,  col: 7).getRow());
    assertEquals( expected: 7, chessBoard.getPiece( row: 0,  col: 7).getColumn());

    assertEquals( expected: 9, chessBoard.getPiece( row: 9,  col: 1).getRow());
    assertEquals( expected: 1, chessBoard.getPiece( row: 9,  col: 1).getColumn());

    assertEquals( expected: 9, chessBoard.getPiece( row: 9,  col: 7).getRow());
    assertEquals( expected: 7, chessBoard.getPiece( row: 9,  col: 7).getColumn());
}
```

Test 9.18: testElephantPosition() (PASSED)
Description: this test gets the row and column of Elephants on the table and compares it with the row and column defined by the programmer in assertEquals()

```
// compare the side of all the elephants on the table to the correct positions
@Test
public void testElephantPosition() {
    // initialize the complete chess board
    ChessBoard chessBoard = init( input: "XIANQI");

    assertEquals( expected: 0, chessBoard.getPiece( row: 0,  col: 2).getRow());
    assertEquals( expected: 2, chessBoard.getPiece( row: 0,  col: 2).getColumn());

    assertEquals( expected: 0, chessBoard.getPiece( row: 0,  col: 6).getRow());
    assertEquals( expected: 6, chessBoard.getPiece( row: 0,  col: 6).getColumn());

    assertEquals( expected: 9, chessBoard.getPiece( row: 9,  col: 2).getRow());
    assertEquals( expected: 2, chessBoard.getPiece( row: 9,  col: 2).getColumn());

    assertEquals( expected: 9, chessBoard.getPiece( row: 9,  col: 6).getRow());
    assertEquals( expected: 6, chessBoard.getPiece( row: 9,  col: 6).getColumn());
}
```

Test 9.19: testRookPosition() (PASSED)
Description: this test gets the row and column of Rooks on the table and compares it with the row and column defined by the programmer in assertEquals()

```
// compare the label of all the rooks on the table to the correct position
@Test
public void testRookPosition() {
    // initialize the complete chess board
    ChessBoard chessBoard = init( input: "XIANQI");

    assertEquals( expected: 0, chessBoard.getPiece( row: 0,  col: 0).getRow());
    assertEquals( expected: 0, chessBoard.getPiece( row: 0,  col: 0).getColumn());

    assertEquals( expected: 0, chessBoard.getPiece( row: 0,  col: 8).getRow());
    assertEquals( expected: 8, chessBoard.getPiece( row: 0,  col: 8).getColumn());

    assertEquals( expected: 9, chessBoard.getPiece( row: 9,  col: 0).getRow());
    assertEquals( expected: 0, chessBoard.getPiece( row: 9,  col: 0).getColumn());

    assertEquals( expected: 9, chessBoard.getPiece( row: 9,  col: 8).getRow());
    assertEquals( expected: 8, chessBoard.getPiece( row: 9,  col: 8).getColumn());
}
```

Test 9.20: testCannonPosition() (PASSED)
Description: this test gets the row and column of Cannons on the table and compares it with the row and column defined by the programmer in assertEquals()

```
// compare the side of all the cannons on the table to the correct position
@Test
public void testCannonPosition() {
    // initialize the complete chess board
    ChessBoard chessBoard = init( input: "XIANQI");

    assertEquals( expected: 2, chessBoard.getPiece( row: 2,  col: 1).getRow());
    assertEquals( expected: 1, chessBoard.getPiece( row: 2,  col: 1).getColumn());

    assertEquals( expected: 2, chessBoard.getPiece( row: 2,  col: 7).getRow());
    assertEquals( expected: 7, chessBoard.getPiece( row: 2,  col: 7).getColumn());

    assertEquals( expected: 7, chessBoard.getPiece( row: 7,  col: 1).getRow());
    assertEquals( expected: 1, chessBoard.getPiece( row: 7,  col: 1).getColumn());

    assertEquals( expected: 7, chessBoard.getPiece( row: 7,  col: 7).getRow());
    assertEquals( expected: 7, chessBoard.getPiece( row: 7,  col: 7).getColumn());
}
```

Test 9.21: testGuardPosition() (PASSED)
Description: this test gets the row and column of Guards on the table and compares it with the row and column defined by the programmer in assertEquals()

```
// compare the label of all the guards on the table to the correct position
@Test
public void testGuardPosition() {
    // initialize the complete chess board
    ChessBoard chessBoard = init( input: "XIANQI");

    assertEquals( expected: "G", chessBoard.getPiece( row: 0,  col: 3).getLabel());
    assertEquals( expected: "G", chessBoard.getPiece( row: 0,  col: 5).getLabel());
    assertEquals( expected: "G", chessBoard.getPiece( row: 9,  col: 3).getLabel());
    assertEquals( expected: "G", chessBoard.getPiece( row: 9,  col: 5).getLabel());
}
```

Test 9.22: testXianqiKingPosition() (PASSED)
Description: this test gets the row and column of Kings on the table and compares it with the row and column defined by the programmer in assertEquals()

```
// compare the label of all the xiangi kings on the table to the correct position
@Test
public void testXiangiKingPosition() {
    // initialize the complete chess board
    ChessBoard chessBoard = init( input: "XIANQI");

    assertEquals( expected: 0, chessBoard.getPiece( row: 0,  col: 4).getRow());
    assertEquals( expected: 4, chessBoard.getPiece( row: 0,  col: 4).getColumn());

    assertEquals( expected: 9, chessBoard.getPiece( row: 9,  col: 4).getRow());
    assertEquals( expected: 4, chessBoard.getPiece( row: 9,  col: 4).getColumn());
}
```

Test 9.23: testChessPieceConstructor() (PASSED)

Description: this test creates a new piece for all kinds of pieces and compare it with its board as well as label

- Test soldier constructor:

```
// test soldier constructor
SoldierPiece soldier = new SoldierPiece(north, chessBoard);
assertEquals(north, soldier.getSide());
assertEquals( expected: "S", soldier.getLabel());
```

- Test horse constructor

```
// test horse constructor
HorsePiece horse = new HorsePiece(north, chessBoard);
assertEquals(north, horse.getSide());
assertEquals( expected: "H", horse.getLabel());
```

- Test elephant constructor

```
// test elephant constructor
ElephantPiece elephant = new ElephantPiece(north, chessBoard);
assertEquals(north, elephant.getSide());
assertEquals( expected: "E", elephant.getLabel());
```

- Test rook constructor

```
// test rook constructor
RookPiece rook = new RookPiece(south, chessBoard);
assertEquals(south, rook.getSide());
assertEquals( expected: "R", rook.getLabel());
```

- Test cannon constructor

```
// test cannon constructor
CannonPiece cannon = new CannonPiece(south, chessBoard);
assertEquals(south, cannon.getSide());
assertEquals( expected: "C", cannon.getLabel());
```

- Test guard constructor

```
// test guard constructor
GuardPiece guard = new GuardPiece(west, chessBoard);
assertEquals(west, guard.getSide());
assertEquals( expected: "G", guard.getLabel());
```

- Test xianqi king constructor

```
// test xianqi king constructor
XianqiKingPiece king = new XianqiKingPiece(east, chessBoard);
assertEquals(east, king.getSide());
assertEquals( expected: "X", king.getLabel());
```

Test 9.24: testBoard() (PASSED)
Description: this test creates a new board for all kinds of board sides and compares with the correct side defined

- Test north board:

```
// test north board
SoldierPiece northBoard = new SoldierPiece(north, chessBoard);
assertEquals(north, northBoard.getSide());
```

- Test south board:

```
// test south board
SoldierPiece southBoard = new SoldierPiece(south, chessBoard);
assertEquals(south, southBoard.getSide());
```

- Test east board:

```
// test east board
SoldierPiece eastBoard = new SoldierPiece(east, chessBoard);
assertEquals(east, eastBoard.getSide());
```

- Test north board:

```
// test west board
SoldierPiece westBoard = new SoldierPiece(west, chessBoard);
assertEquals(west, westBoard.getSide());
```

Test 9.25 testFirstPlayer() (PASSED)
Description: I created a new method in the ChessGame interface to get the first player. This test will return true if the first player's side matches the programmer's input.

- Test north being the first player:

```
// test north being the first player
ChessBoard northBoard = init( input: "XIANQI", north);
assertEquals(north, northBoard.getGameRules().getFirstPlayer());
```

- Test south being the first player:

```
// test south being the first player
ChessBoard southBoard = init( input: "CHESS", south);
assertEquals(south, southBoard.getGameRules().getFirstPlayer());
```

- Test east being the first player:

```
// test east being the first player
ChessBoard eastBoard = init( input: "XIANQI", east);
assertEquals(east, eastBoard.getGameRules().getFirstPlayer());
```

- Test west being the first player:

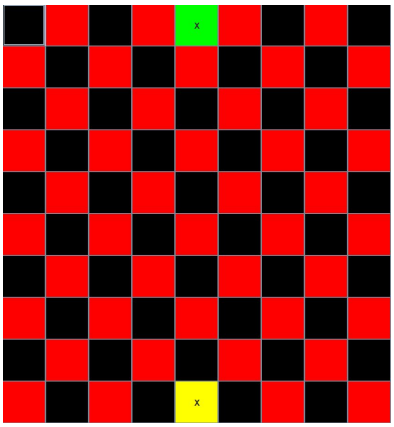
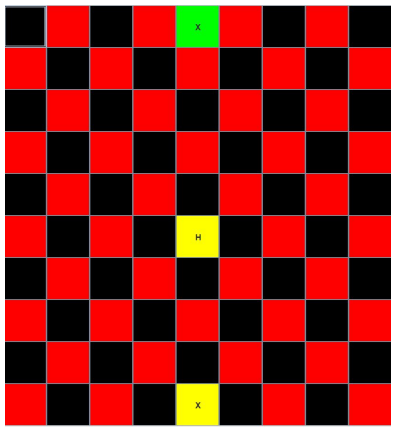
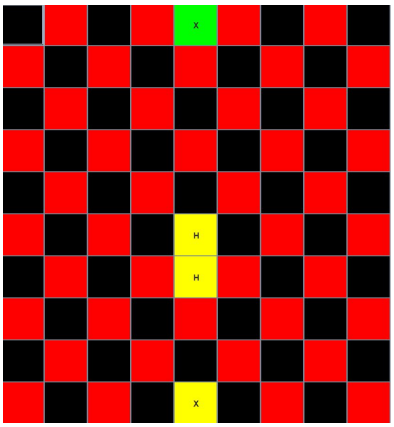
```
// test west being the first player
ChessBoard westBoard = init( input: "XIANQI", west);
assertEquals(west, westBoard.getGameRules().getFirstPlayer());
```

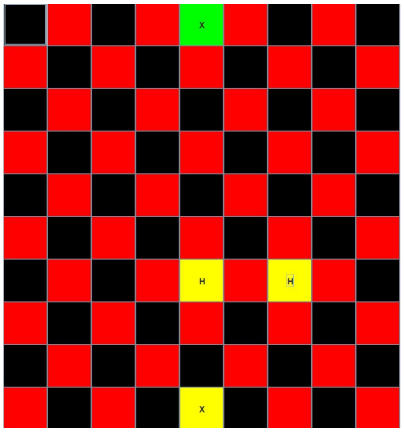
10)TEST FACING KING

Note: We will test with the terminal and some moves when the king is facing or the king is not or a move that can trigger a facing king. We will set the facingKing method to be static so that we can test in Xianqi's main method.

Description:

- In the first picture, two Kings are facing each other, making the value in the terminal appear true.
- In the first picture, two Kings are blocked by a Horse, making the value in the terminal appear false. However, we cannot move the Horse because it will trigger two kings facing each other.
- In the first picture, two Kings are blocked by 2 Horses, making the value in the terminal appear false. In this case, we can move one of the Horses because it will not trigger two kings facing each other since there is still one Horse blocking 2 Kings.

		
Value in terminal: **Facing Kings: true**	Value in terminal: **Facing Kings: false**	Value in terminal: **Facing Kings: false**



11) BUGS AND SOLUTIONS SO FAR

Description:

In my process of coding, I faced some bugs, mostly related to checking whether a castle move is valid or not using the ***squareThreatened()*** method. After debugging, I found out that the error was StackOverflowError, which resulted from recursion logic I made in my code while trying to put ***squareThreatened()*** into ***legalCastle()*** in ***SPiece.java***.

Here is the analysis of my problem:

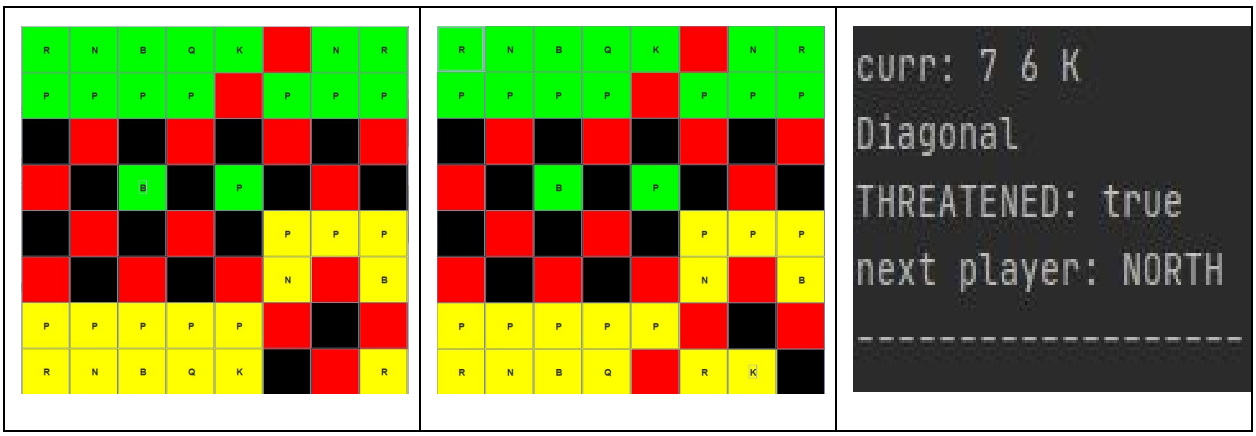
squareThreatened=>isLegalMovecalled=>legalKing=>legalCastle=>squareThreatened

To solve this problem, I decided to not use ***squareThreatened()*** in ***legalCastle()*** anymore. Therefore, I decided to check my castle move in ***makeMove()*** in ***EuropeanChess.java***. By creating a new method called ***castleThreatened()***, I managed to check all possibilities of my castle being attacked or not.

Eventually, if I tested my method individually, which I put in specific cases, the result came out true. But when I tried to generalize my code, it made my program run weird.

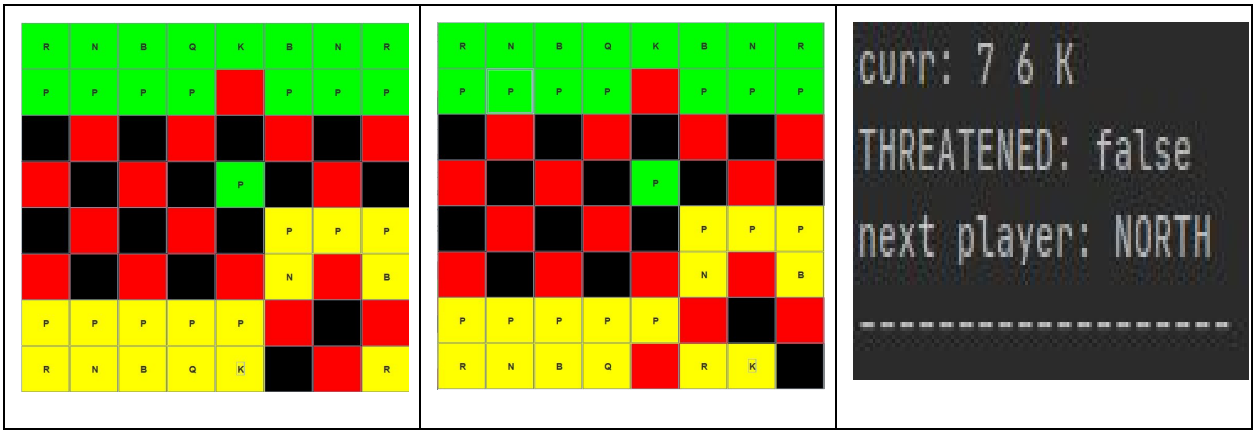
Here is a statistic of my code when I put in a specific case the Bishop of team North attacking the Castle Move of team Yellow:

```
if (piece.getChessBoard().hasPiece( row: 3, col: 2)
        && piece.getChessBoard().getPiece( row: 3,  col: 2).getSide() != piece.getSide() &&
        piece.getChessBoard().getPiece( row: 3,  col: 2).isLegalMove(toRow,  toColumn: 6)) {
    return true;
}
```



Values in the terminal:

- ***curr:*** the current position of the king is (7, 6)
- ***K:*** labeled for King, which tells us that the King is making the move
- ***THREATENED:*** the position of (7, 6) is currently being attacked
- ***next player:*** the side of the next player taking the turn



Values in the terminal:

- ***curr:*** the current position of the king is (7, 6)
- ***K:*** labeled for King, which tells us that the King is making the move
- ***THREATENED:*** the position of (7, 6) is currently not being attacked
- ***next player:*** the side of the next player taking the turn

We can see that in two cases I tested above with the Bishop attacking the square (7, 6), it seems that the result came out true according to the THREATENED value.

I'm still debugging the code by the time I submitted all my files. If you have any suggestions to this problem, can you please let me know?

12)SOME FINDINGS IN THE CODE

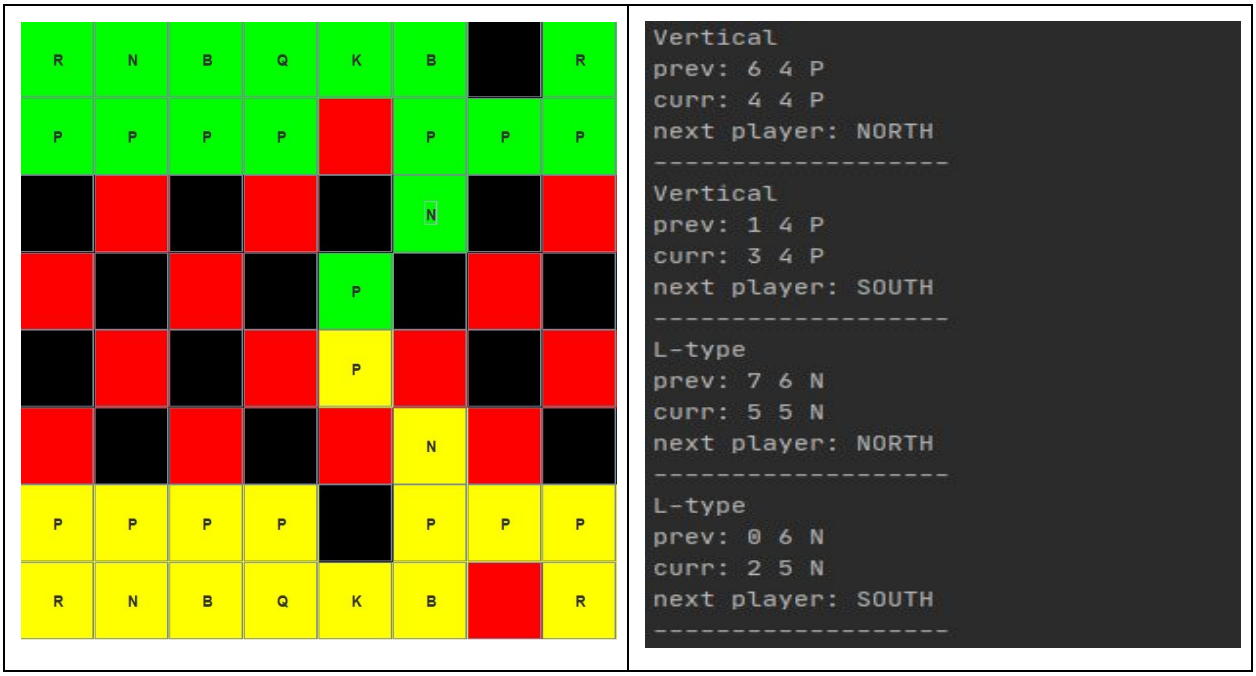
canChangeSelection():

I found out in my code that we can actually set the canChangeSelection to be always **true** because a move is either an illegalMove or a legalMove. If a legalMove is made, the game will automatically change its chessBoard. Otherwise, no move is made. If you click on an empty space, the game will do nothing. Also, the legalPieceToPlay also decides for you all the possible pieces that you can play.

upgradePawn():

Provide a default shortens the String processing code. In this case, we need to provide something for the label if String is null.

makeMove():



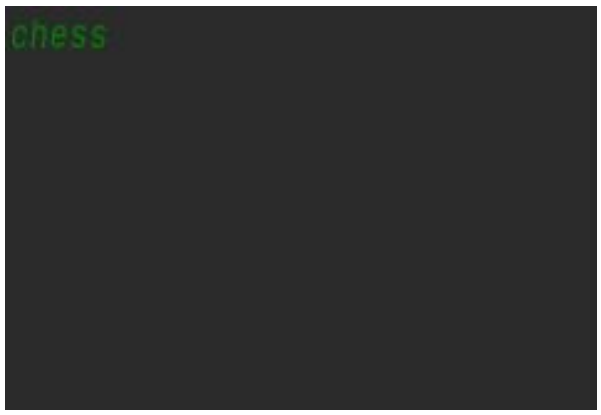
JAVAFX BOARD

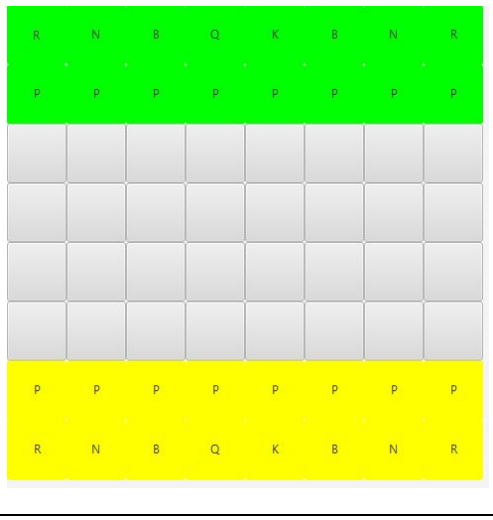
Note: We will only test several game states of this javafx because the logic is tested above. So the only thing needed to be considered is how the game displays and how the game is played.

1) **Display**

1.1) European Chess

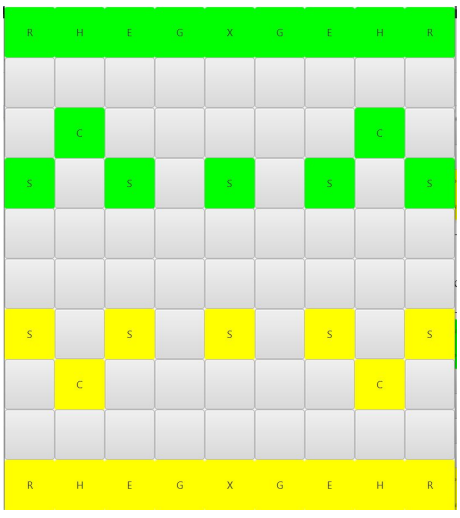
Description: We will first type in the terminal the type of game we want to play, and then it will display the game, which is european chess.





1.2) Xianqi

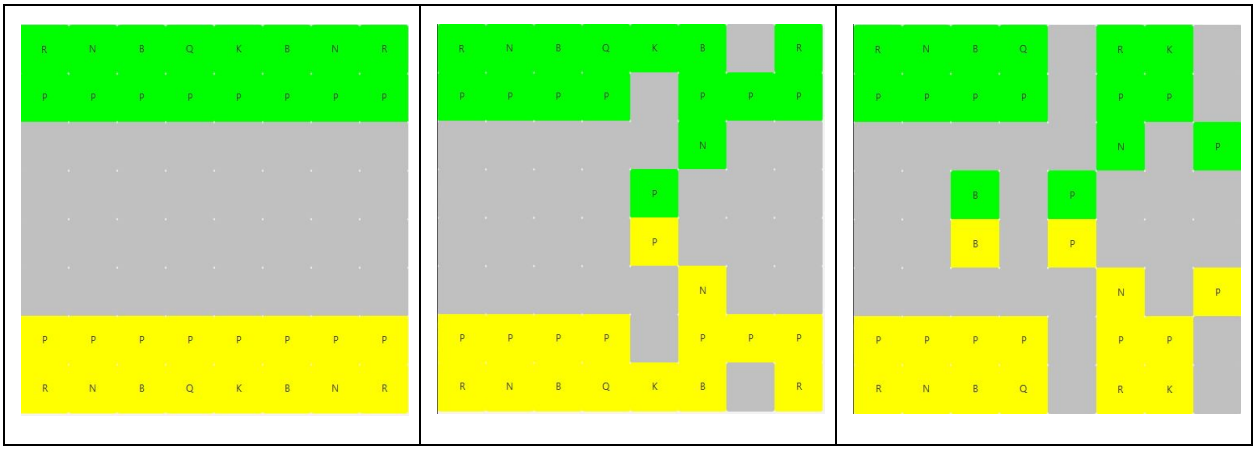
Description: We will first type in the terminal the type of game we want to play, and then it will display the game, which is xianqi.



2) Game Play

2.1) European Chess

Description: I will print out several game states of me playing with myself since the logic is tested above.

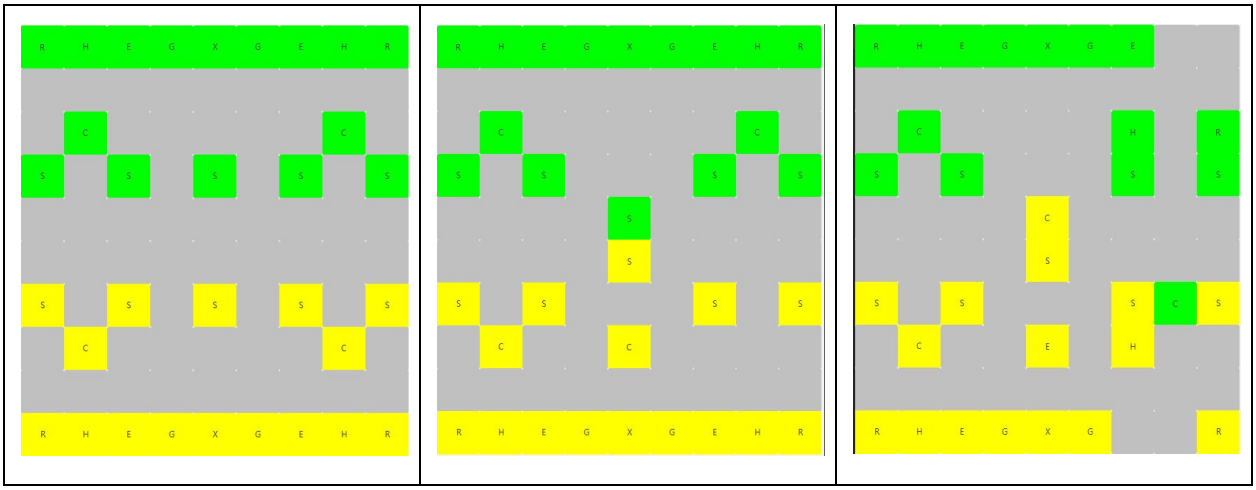


Here is what is being printed out in the back while you are playing the game (with order of moves played) from picture 1 to 3.

1) Vertical prev: 6 4 P curr: 4 4 P next player: NORTH 2) Vertical prev: 1 4 P curr: 3 4 P next player: SOUTH 3) L-type prev: 7 6 N curr: 5 5 N next player: NORTH	4) L-type prev: 0 6 N curr: 2 5 N next player: SOUTH 5) Diagonal prev: 7 5 B curr: 4 2 B next player: NORTH 6) Diagonal prev: 0 5 B curr: 3 2 B next player: SOUTH	7) Diagonal prev: 7 6 K curr: 7 6 K Diagonal THREATENED: false next player: NORTH 8) Diagonal prev: 0 6 K curr: 0 6 K THREATENED: false next player: SOUTH	9) Vertical prev: 6 7 P curr: 5 7 P next player: NORTH 10) Vertical prev: 1 7 P curr: 2 7 P next player: SOUTH

2.2) Xianqi

Description: I will print out several game states of me playing with myself since the logic is tested above.



Here is what is being printed out in the back while you are playing the game (with order of moves played) from picture 1 to 3.

1) Vertical prev: 6 4 S curr: 5 4 S next player: NORTH 2) Vertical prev: 3 4 S curr: 4 4 S next player: SOUTH 3) Horizontal prev: 7 7 C curr: 7 4 C next player: NORTH	4) Vertical prev: 7 4 C curr: 4 4 C next player: SOUTH 5) Vertical prev: 2 7 C curr: 6 7 C next player: NORTH 6) L-type prev: 9 7 H curr: 7 6 H next player: SOUTH	7) L-type prev: 0 7 H curr: 2 6 H next player: NORTH 8) Diagonal prev: 9 6 E curr: 7 4 E next player: SOUTH 9) Diagonal prev: 9 6 E curr: 7 4 E next player: SOUTH

3) Button Responsiveness

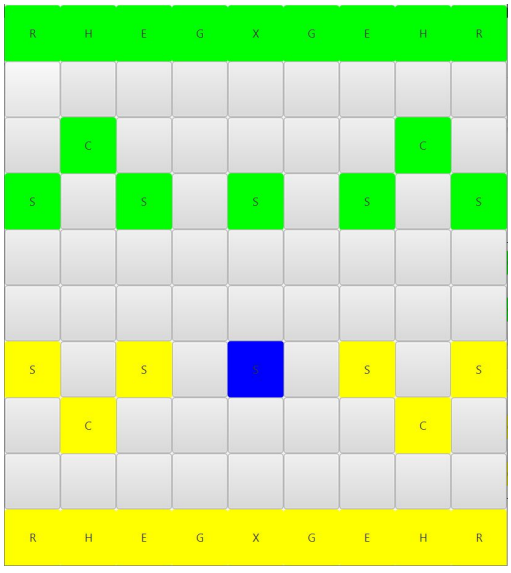
3.1) Handle Buttons

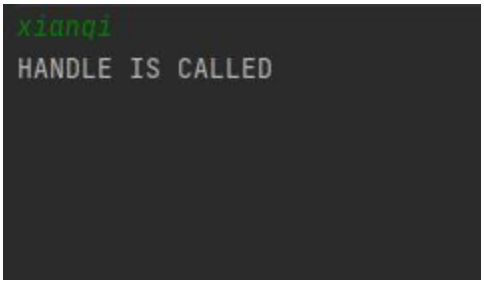
Description: We have the handle method here, each time the button is clicked, it will show up that it is called.

```
@Override
public void handle(ActionEvent e) {
    System.out.println("HANDLE IS CALLED");
    Button b = (Button) e.getSource();
    int col = -1;
    int row = -1;

    // first find which button (board square) was clicked.
    for (int i = 0; i < squares.length; i++) {
        for (int j = 0; j < squares[i].length; j++) {
            if (squares[i][j].equals(b)) {
                row = i;
                col = j;
            }
        }
    }

    if (firstPick) {
        processFirstSelection(row, col);
    }
    else {
        processSecondSelection(row, col);
    }
}
```

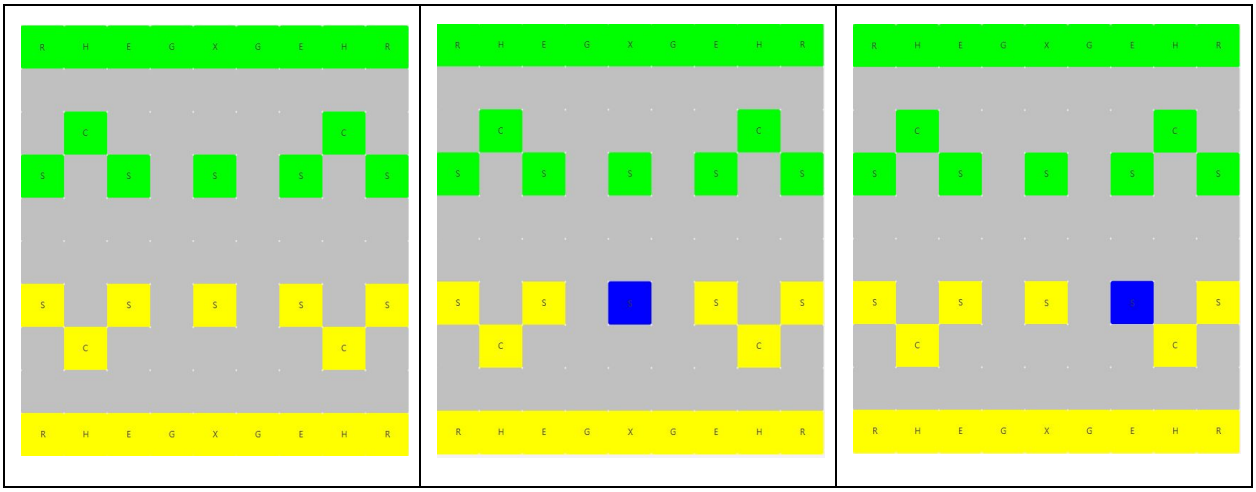




Appear in the terminal

3.2) Change Selection

Description: We will test whether the board allows us to change our selection.



BONUS: Here are some random pictures of the game I coded