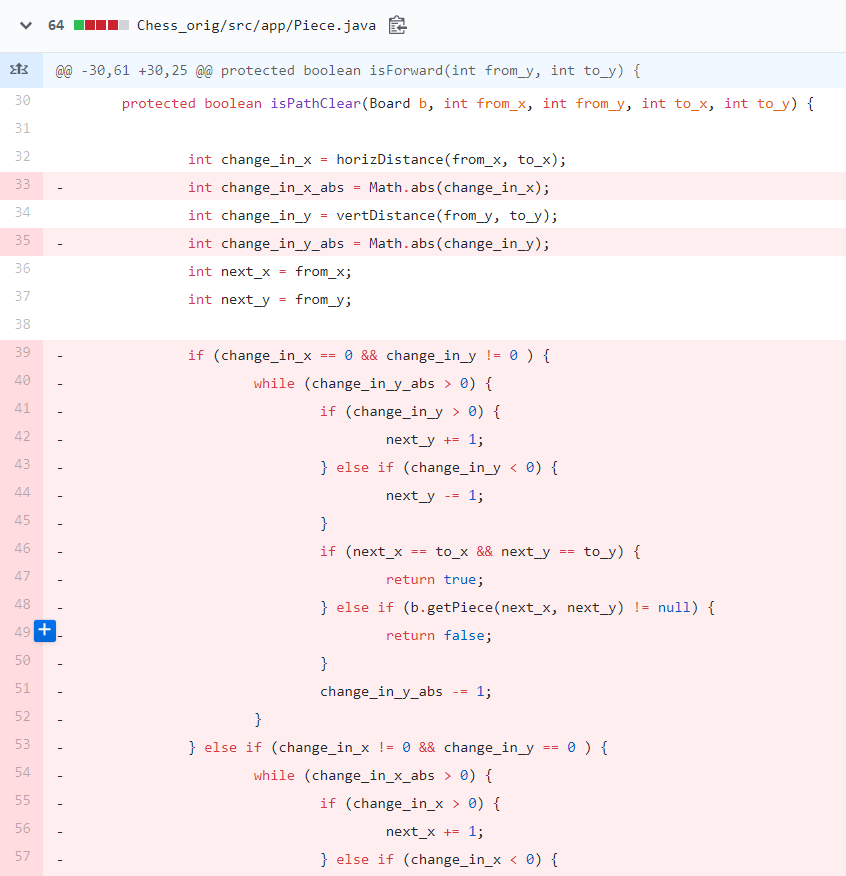
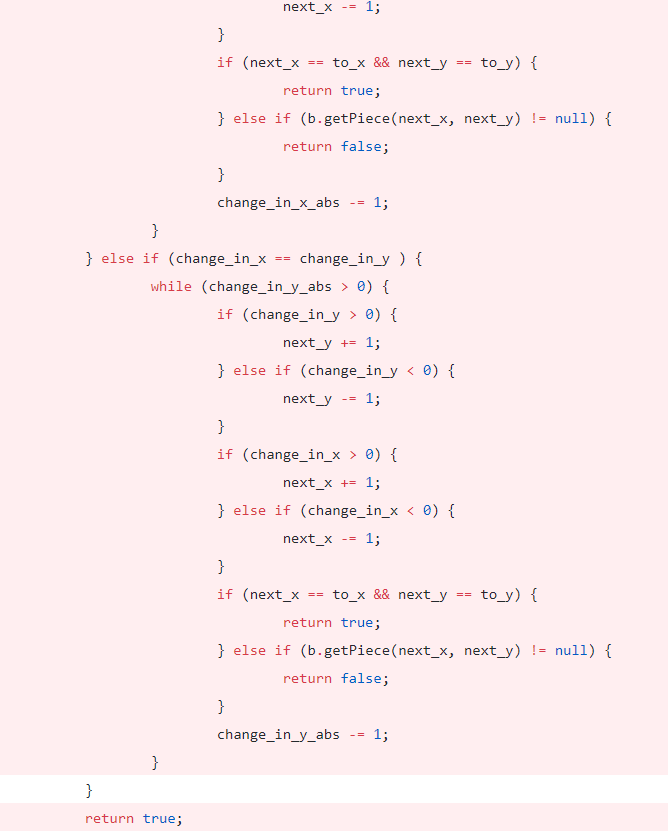
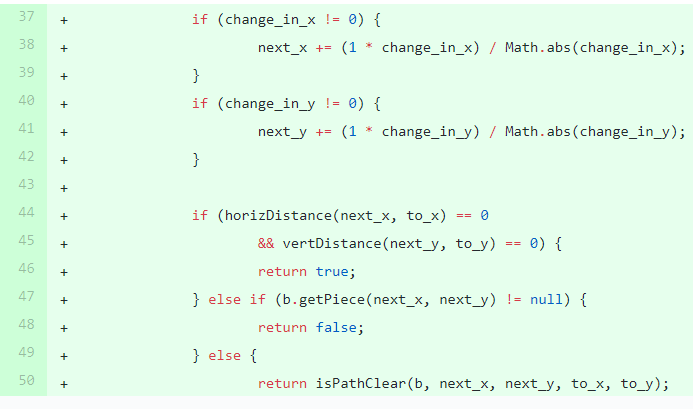
**CPSC ASSIGNMENT 1 REPORT**

Refactoring 1

This refactoring changed the code of the protected “isPathClearMethod” of the abstract Piece class. This method is used by all its subclasses except for the King and Knight class.

The method was not working properly as detected by: testBishopCanMoveDiagonalMoveUpRightPathNotClear() and testBishopCanMoveDiagonalMoveDownLeftPathNotClear() methods in the BishopTest JUnit test case as well as the equivalent methods in the QueenTest JUnit test case. The method also exhibited “Duplicate Code” and “Long Method” code smells.

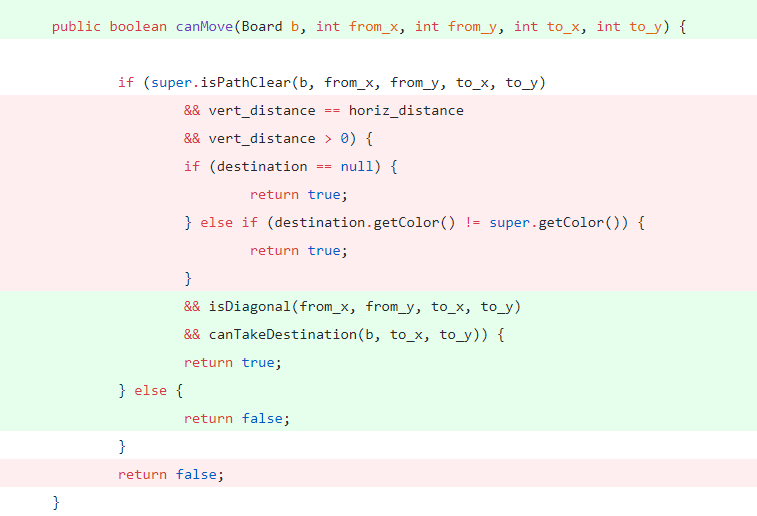
I applied the “Substitute Algorithm” refactoring by ensuring that the prepared alternative algorithm compiled successfully, followed by running the already prepared “…PathNotClear()” JUnit test cases for all Piece subclasses that are able to move multiple squares and cannot skip over pieces. The newly improved algorithm passed all tests successfully.

The result of the refactoring is the “isPathClearMethod()” in the abstract Piece class.

Because the “isPathClearMethod()” is protected, it was tested by running JUnit test cases on the canMove() method of the Piece subclasses that can move multiple pieces in a direction, without being able to skip over other pieces. For each direction one of these pieces can move, a Board object would be initialized with a different piece placed in the way of movement. The canMove() method of these pieces will then call isPathClearMethod() and return false if detecting the piece along the path of movement.

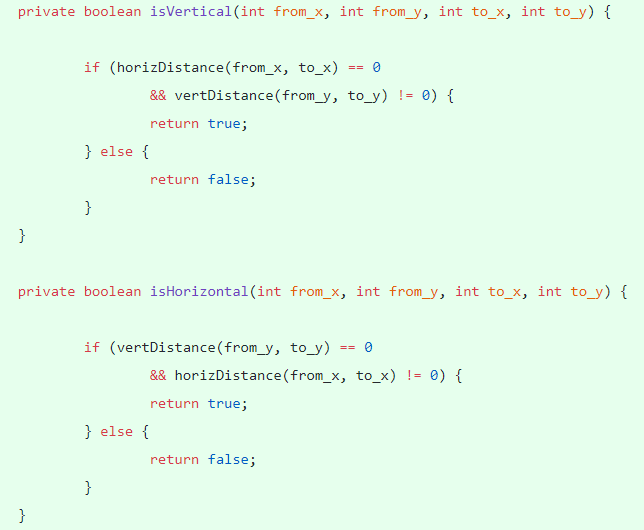
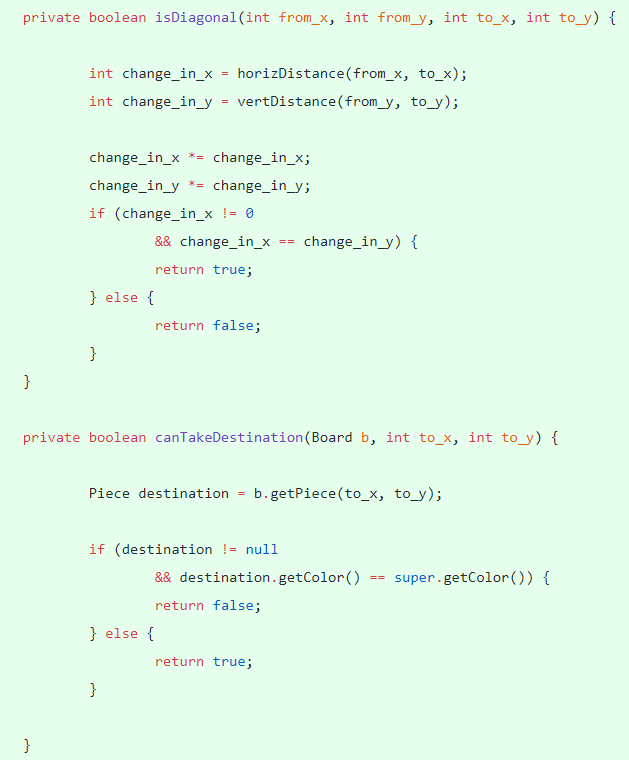
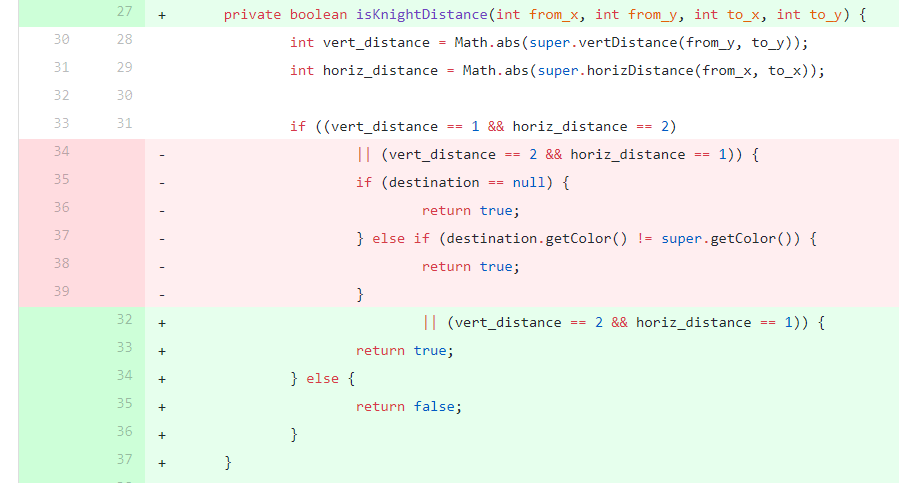
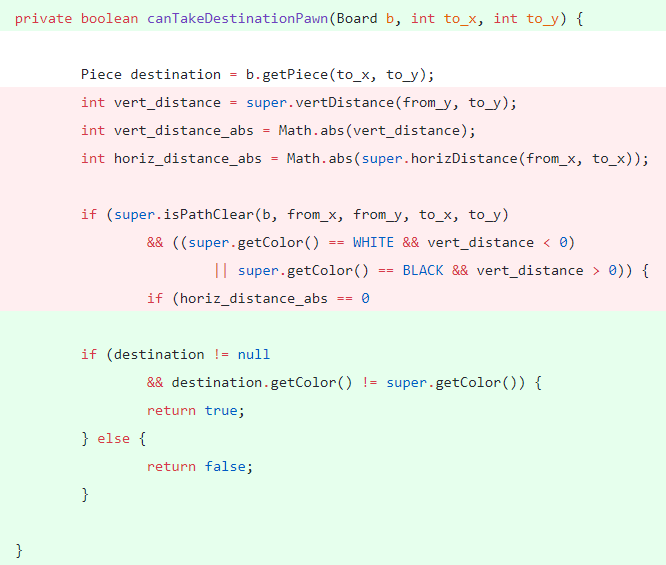
Aside from now successfully being able to tell when a path is clear for diagonal movement, the new algorithm is also much shorter, due to less code duplication, and easier to understand, due to previous nested conditionals with while loops being removed.

Refactoring 2

This refactoring changed the code of the canMove() method. This method is an abstract method of the Piece.java class, it is implemented by all its subclasses: Bishop.java, King.java. Pawn.java, Rook.java, Knight.java, Queen.java.

The method previously had the long method code smell. Complex conditionals made the flow of control difficult to read.

I applied the Decompose Conditional refactoring via Extract Method, the newly created methods are then called instead in canMove().

New method isDiagonal, canTakeDestination added to Bishop.java, new method isHorizontal, isVertical, new method isDiagonal, canTakeDestination added to King.java, new method isKnight, canTakeDestination added to Knight.java, new method isVertical, isDiagonal, canTakeDestinationPawn added to Pawn.java, new method isVertical, isHorizontal, isDiagonal, canTakeDestination added to Queen class, new method isVertical, isHorizontal, canTakeDestination added to Rook class.

The newly added methods are all private as they are helper methods and thus they are tested through running JUnit tests on the canMove() method of each respective class.

The refactoring resulted in the canMove() methods being much cleaner and easier to read. However, it also led to a large amount of code duplication enabling further refactorings such as the pull up method.

Refactoring 3

This refactoring changed new code added as a result of Refactoring 2.

Previous to the refactoring there were many classes implementing the same methods resulting in the code duplication code smell.

I applied the Pull Up Method, by creating a new method in the superclass, copying the body of a method to it, adjusting for any fields that were only visible in the subclass, compiling and testing until only the superclass method remained.

The isVertical(), isHorizontal(), isDiagonal(), canTakeDestination() implemented in the abstract piece class resulted in the refactoring.

The methods pulled up are tested in the same way they were tested in the previous refactoring.

After the refactoring there is less lazy repetitive code, if the method needs to be changed now it is in a single place rather than having to search for all duplicates of the same method.

Refactoring 4

This refactoring removed the code of the isForward() method implemented in Piece.java.



The Refused Bequest code smell was detected as previous to the refactoring only a single subclass was using the method implemented in the superclass.

The push down refactoring was used for this refactoring. I declared the new method in all subclasses and then copied the body over. After this I removed the method from the superclass, compiled and tested. Then I removed the method from each subclass that did not need it, and compiled and tested again.

As a result of this refactoring the isForward() method was added to Pawn.java.

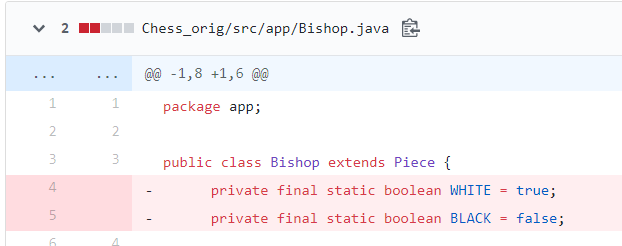


The code was tested by running JUnit test cases on canMove() method of Pawn.java which then in turn invokes isForward().

The result of this refactoring is improved class coherence. The method is now located where one would expect to see it.

Refactoring 5

Constants are removed from Piece.java’s subclasses: Bishop.java, King.java. Pawn.java, Rook.java, Knight.java, Queen.java.



Each class was using the same constants, creating a code duplication smell.

I used the Pull Up Field refactoring, making the constants in the Piece class protected instead of private, and removing them in its subclasses.

As a result of the refactoring the constants in Piece.java are changed from being private to being protected.



The code was tested by running the same JUnit tests to see if any functionality had changed.

There is less code duplication resulting in any changes to the constants being easier to implement as there they do not need to be changed across multiple classes.