As a team we decided to do a Walkthrough of the code used as an algorithm to look for check in our Chess program. We decided to do this as we had just finished up work on this section of code and thought it would be pertinent to check code line-by-line to make sure that such a crucial aspect of our program is functional. This activity was performed by all group members. Everyone was present as the person who coded this segment of code described what each line did.

To detect whether the king is in check, the program first finds the king that is on the current player's side of the board. This happens in the GameBoard method startTurn, which is triggered when a piece is moved and when a piece is clicked. It basically sets up all the appropriate buttons for the turn. After all the appropriate buttons are enabled, another nested loop looks for the currently activated king by going through all pieces and filtering by color and type. When the king is found, it is passed to the CheckFind method along with the current color. This function iterates through the board again and searches for any pieces that could attack the king. It does this by using the couldAttack methods. These methods belong to every kind of piece (including, trivially, for null). However, they are still unique for each piece. They all take in a piece as a parameter, and return true if the parameter piece could potentially be attacked by the piece to which the method belongs. They return false otherwise. In CheckFind, all of the pieces that could attack the king of the current color are pushed to an array. This is mainly an artifact of the initial plan to enable every button for every piece that would be able to attack a piece that was holding the king in check. This array is returned to startTurn. If the array is not empty, this means that the king could be attacked on the next turn, so startTurn calls another method, buttonsConfigCheck. This method takes in the king of the current color as a parameter. As of now, it simply disables every button that is not the king.

The faults we found were mostly trivial. A lot of the mistakes found could be solved by fixing only one line of code, but it was still important that we were able to find these. Without fixing these faults the program would be less stable and its good practice to make sure that even the small issues are dealt with before we turn in the assignment.

The first fault we found was in regards to one of the functions that ran to check if a piece could defend their king by attacking an opposing piece that was threatening it. This function would run for a “null” piece, so in practice it would rarely be seen, but instead of returning false it returned an empty string. We were able to fix it very quickly. This fault was found by Mat and fixed by Zack

The second fault we found was a superfluous return in a nested for loop. The for loop is designed to end when both loops reach their max value, in this case 7 and 7. There is a return command placed in an if statement when both loops are equal to 7 which is in essence redundant. This fault was found by Mat

One of the new functions for this section of code was title buttonConfigCheck(). We all agreed that this naming was obtuse and needed to be fixed. This was also pointed out by Mat

The final issue we found was that the King was able to move into check. In the official rules of chess this is an illegal move and is directly involved with this section of our code. This functionality was already planned on being implemented but it was important to note that is was missing. This was brought up by Ian.