Criterion C – Product Development

Techniques Used:

Additional Information:

Appendix 1 – evidence of development

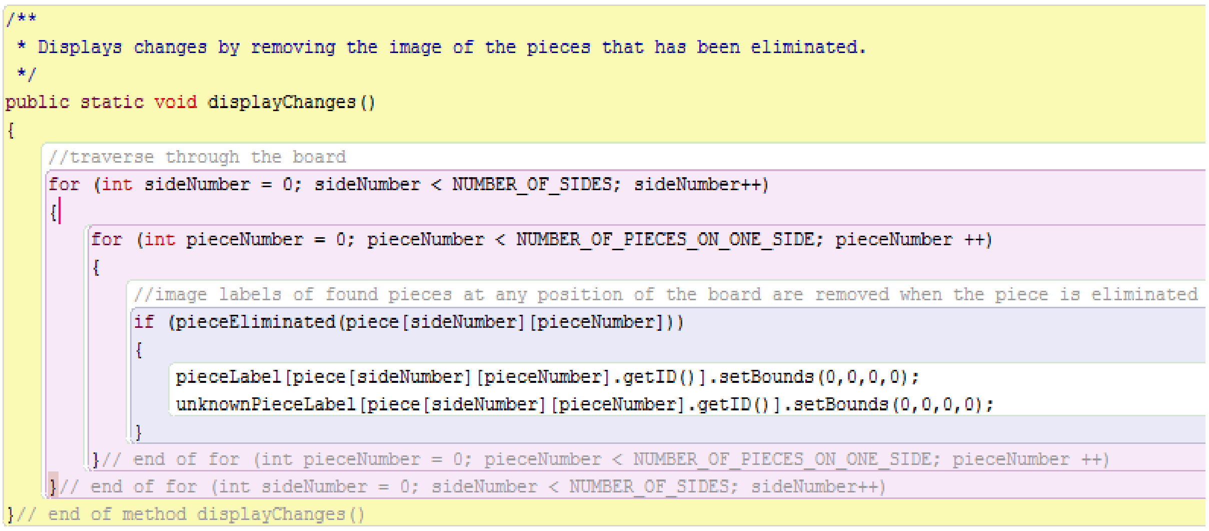
Appendix 2 – bibliography

Graphical display

JLabels, added to the pane of a JFrame were used to display images. Before the user even receives a welcome message for the game, the 50 pieces were created and so was the 100 JLabels. 50 JLabels were used for the image of every piece and the other 50 for the cover of each piece.

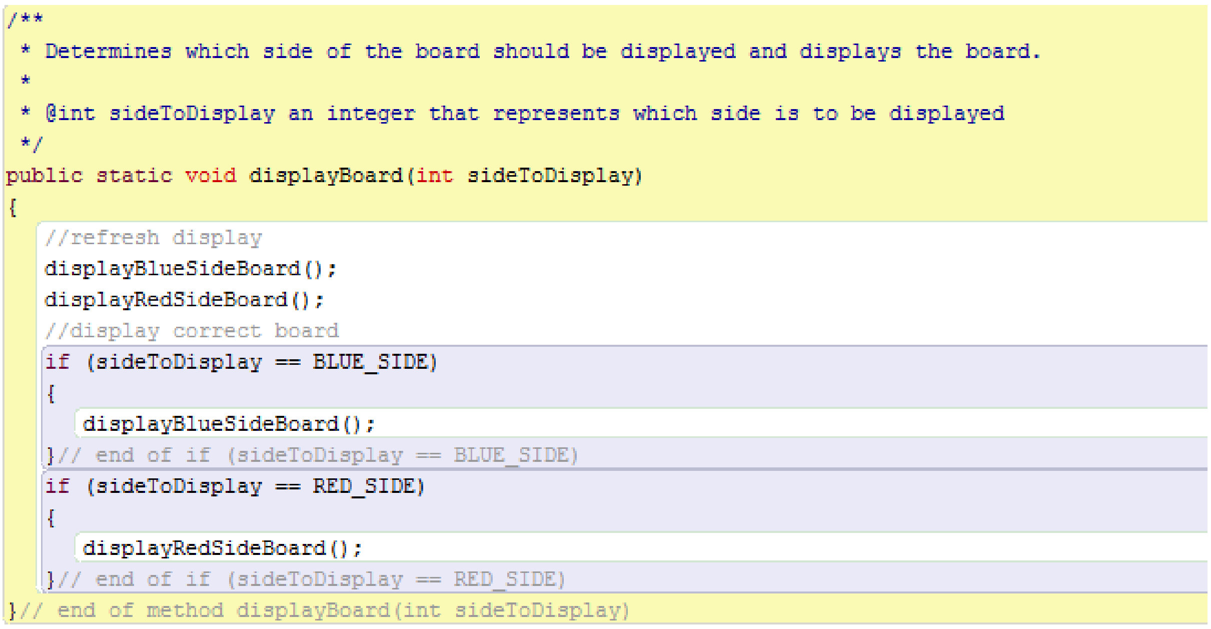
After placing each piece or moving a piece, the image of the board must reflect the changes taking place on the board. On top of that, the display must also take into consideration whose turn it is, so that any opponent pieces remain covered.

The displayChanges() method is called after every turn, so that any eliminated pieces no longer appear on the board. With the image of the piece itself, the image cover is also removed.



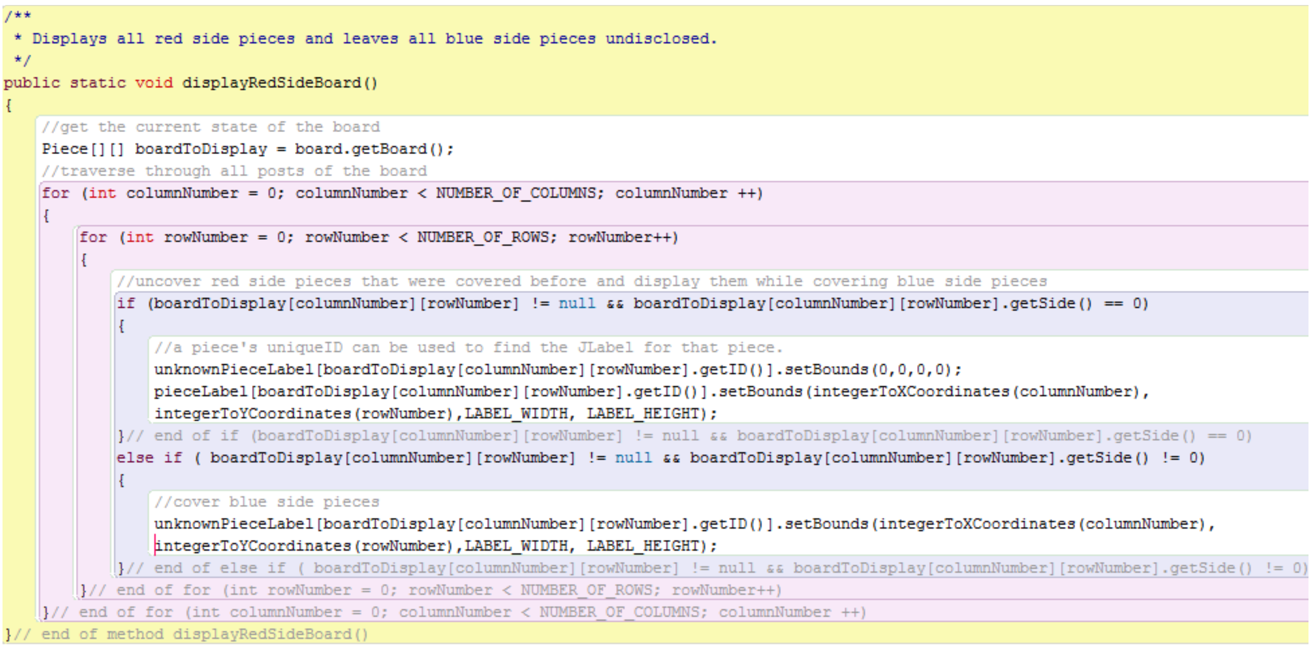
```
/**
 * Displays changes by removing the image of the pieces that has been eliminated.
 */
public static void displayChanges()
{
    //traverse through the board
    for (int sideNumber = 0; sideNumber < NUMBER_OF_SIDES; sideNumber++)
    {
        for (int pieceNumber = 0; pieceNumber < NUMBER_OF_PIECES_ON_ONE_SIDE; pieceNumber ++)
        {
            //image labels of found pieces at any position of the board are removed when the piece is eliminated
            if (pieceEliminated(piece[sideNumber][pieceNumber]))
            {
                pieceLabel[piece[sideNumber][pieceNumber].getID()].setBounds(0,0,0,0);
                unknownPieceLabel[piece[sideNumber][pieceNumber].getID()].setBounds(0,0,0,0);
            }
        }// end of for (int pieceNumber = 0; pieceNumber < NUMBER_OF_PIECES_ON_ONE_SIDE; pieceNumber ++)
    }// end of for (int sideNumber = 0; sideNumber < NUMBER_OF_SIDES; sideNumber++)
}// end of method displayChanges()
```

Still, after the removal of eliminated pieces, other pieces could have still moved and the board still needs to be re-displayed. For example, when an attacking Field Marshall collides with a Lieutenant, the Lieutenant is eliminated but the field marshall also moves to the place of the lieutenant. Therefore, after each turn or the placement of a piece, displayBoard(int sideNumber) is called.



```
/**
 * Determines which side of the board should be displayed and displays the board.
 *
 * @int sideToDisplay an integer that represents which side is to be displayed
 */
public static void displayBoard(int sideToDisplay)
{
   //refresh display
   displayBlueSideBoard();
   displayRedSideBoard();
   //display correct board
   if (sideToDisplay == BLUE_SIDE)
   {
      displayBlueSideBoard();
   }// end of if (sideToDisplay == BLUE_SIDE)
   if (sideToDisplay == RED_SIDE)
   {
      displayRedSideBoard();
   }// end of if (sideToDisplay == RED_SIDE)
}// end of method displayBoard(int sideToDisplay)
```

displayRedSideBoard() is shown below. The only difference between this method and displayBlueSideBoard() is the statement “if this piece’s side is RED” compared to “if this piece’s side is BLUE”



```
/**
 * Displays all red side pieces and leaves all blue side pieces undisclosed.
 */
public static void displayRedSideBoard()
{
   //get the current state of the board
   Piece[][] boardToDisplay = board.getBoard();
   //traverse through all posts of the board
   for (int columnNumber = 0; columnNumber < NUMBER_OF_COLUMNS; columnNumber ++)
   {
      for (int rowNumber = 0; rowNumber < NUMBER_OF_ROWS; rowNumber++)
      {
         //uncover red side pieces that were covered before and display them while covering blue side pieces
         if (boardToDisplay[columnNumber][rowNumber] != null && boardToDisplay[columnNumber][rowNumber].getSide() == 0)
         {
            //a piece's uniqueID can be used to find the JLabel for that piece.
            unknownPieceLabel[boardToDisplay[columnNumber][rowNumber].getID()].setBounds(0,0,0,0);
            pieceLabel[boardToDisplay[columnNumber][rowNumber].getID()].setBounds(integerToXCoordinates(columnNumber),
            integerToYCoordinates(rowNumber),LABEL_WIDTH, LABEL_HEIGHT);
         }// end of if (boardToDisplay[columnNumber][rowNumber] != null && boardToDisplay[columnNumber][rowNumber].getSide() == 0)
         else if ( boardToDisplay[columnNumber][rowNumber] != null && boardToDisplay[columnNumber][rowNumber].getSide() != 0)
         {
            //cover blue side pieces
            unknownPieceLabel[boardToDisplay[columnNumber][rowNumber].getID()].setBounds(integerToXCoordinates(columnNumber),
            integerToYCoordinates(rowNumber),LABEL_WIDTH, LABEL_HEIGHT);
         }// end of else if ( boardToDisplay[columnNumber][rowNumber] != null && boardToDisplay[columnNumber][rowNumber].getSide() != 0)
      }// end of for (int rowNumber = 0; rowNumber < NUMBER_OF_ROWS; rowNumber++)
   }// end of for (int columnNumber = 0; columnNumber < NUMBER_OF_COLUMNS; columnNumber ++)
}// end of method displayRedSideBoard()
```

Both these methods integerToXCoordinates(columnNumber) and integerToYCoordinates(rowNumber) to convert the row on the board to the x coordinates and y coordinates of the screen so they can be displayed by the setBounds() method of the JLabel class. These x and y coordinates of the screen for each column and row and constants. Ex.

private static final int COLUMN_A_COORDINATE = -10;

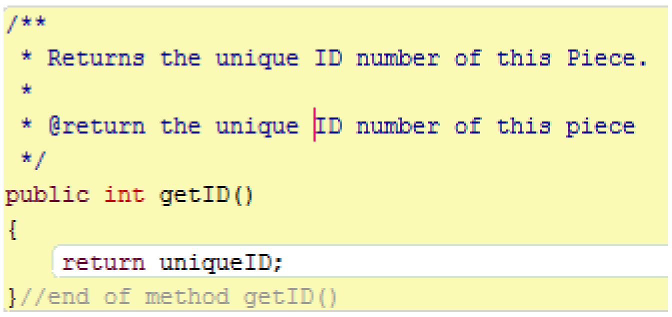
private static final int COLUMN_B_COORDINATE = 95;

private static final int ROW_15_COORDINATE = 506;

Methods

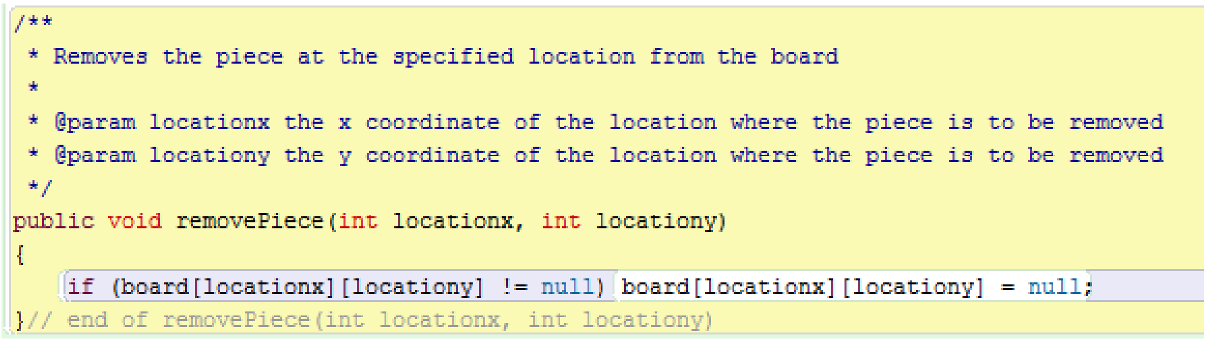
Methods of the Piece and Board class must be called against a Piece or Board respectively. Methods in the Game class are static and do not associate iself with an object. Methods in the Piece and Board class are either accessors or mutators. These methods are employed to make the program more effcient as they can be called every time an object is called against it, instead of rewriting the code multiple times.

Accessor



```java
/**
 * Returns the unique ID number of this Piece.
 *
 * @return the unique ID number of this piece
 */
public int getID()
{
    return uniqueID;
}//end of method getID()
```

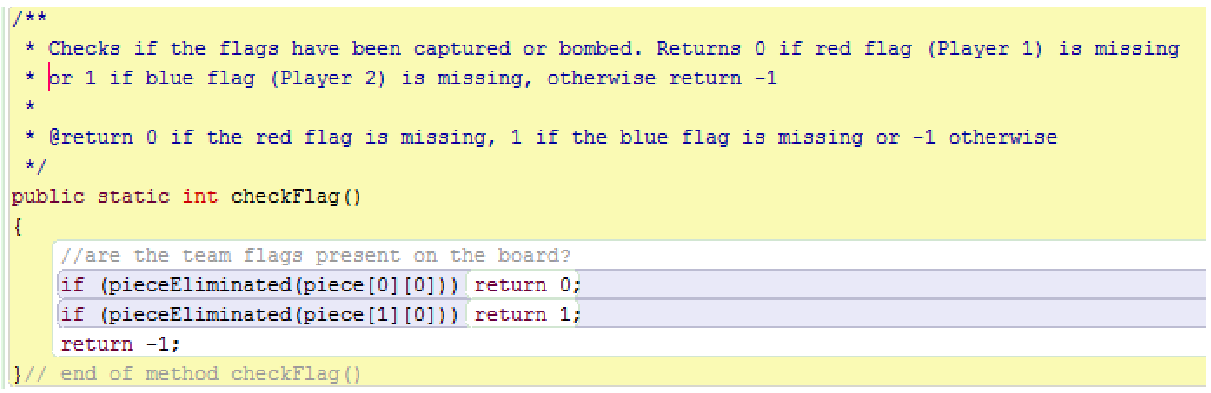
Mutator



```java
/**
 * Removes the piece at the specified location from the board
 *
 * @param locationx the x coordinate of the location where the piece is to be removed
 * @param locationy the y coordinate of the location where the piece is to be removed
 */
public void removePiece(int locationx, int locationy)
{
    if (board[locationx][locationy] != null) board[locationx][locationy] = null;
}// end of removePiece(int locationx, int locationy)
```

Methods of the class Game are called several times by other methods. Most methods perform a specific task such as converting a columnNumber to the x coordinates on the screen. The ancestor of all methods is the main method.

The following method checks if either side’s flag has been eliminated



```java
/**
 * Checks if the flags have been captured or bombed. Returns 0 if red flag (Player 1) is missing
 * or 1 if blue flag (Player 2) is missing, otherwise return -1
 *
 * @return 0 if the red flag is missing, 1 if the blue flag is missing or -1 otherwise
 */
public static int checkFlag()
{
    //are the team flags present on the board?
    if (pieceEliminated(piece[0][0])) return 0;
    if (pieceEliminated(piece[1][0])) return 1;
    return -1;
}// end of method checkFlag()
```

This method is called by the displayWinner() method which is called by the takeTurn() method. takeTurn() is called by the main method.

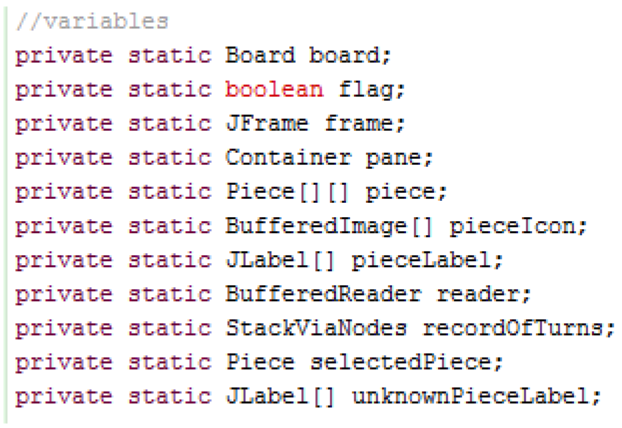
Variables

Local variables include Piece[][] boardToDisplay, local to the displayRedSideBoard() method above. This variable is assigned the state of the current board, it is then used to traverse the board. These variables are employed to store moves, pieces, JLabels and other information required to place pieces and make moves. Other variables include BufferedReader reader which is local to many methods that require input from players, such as takeTurn() and setUpPieces().

Another local variable is inputOK which is used to determine whether the player input is valid. This variable name is used in many methods and are local to only those methods.

Other local variables include currentXCoordinate, currentYCoordinate, destinationXCoordinate, destinationYCoordinate and unknownPieceIcon that are local to the methods. Variables such as sideNumber, columnNumber, rowNumber are often local to a for loop.

Global variables are listed below.



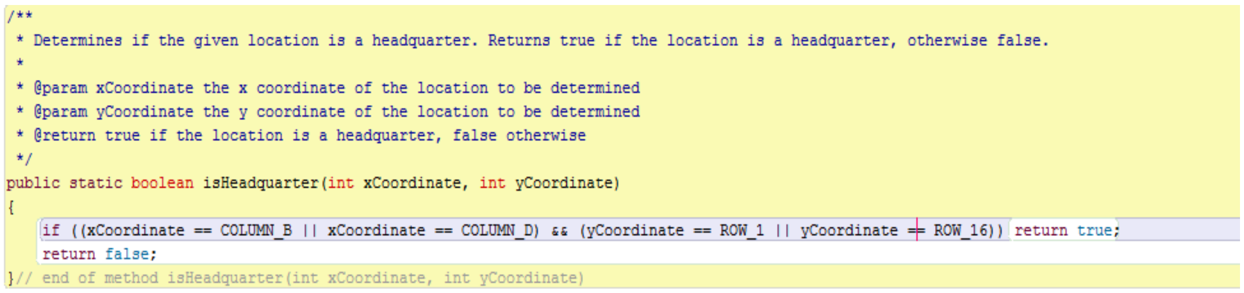
```
//variables
private static Board board;
private static boolean flag;
private static JFrame frame;
private static Container pane;
private static Piece[][] piece;
private static BufferedImage[] pieceIcon;
private static JLabel[] pieceLabel;
private static BufferedReader reader;
private static StackViaNodes recordOfTurns;
private static Piece selectedPiece;
private static JLabel[] unknownPieceLabel;
```

These variables are used throughout the game and called by several methods. **board** holds the current positions of all the pieces. **flag** is needed to keep track of when a flag is captured and the game is over. The variables **frame, pane, unknownPieceLabel, pieceLabel** are required to display the board and the pieces at all times. **piece** is referenced throughout the game as it stores all of the 50 pieces in the game. Finally **recordOfTurns** is a stack that stores each board after it has been changed so that previous boards can be popped and the turn can be undone.

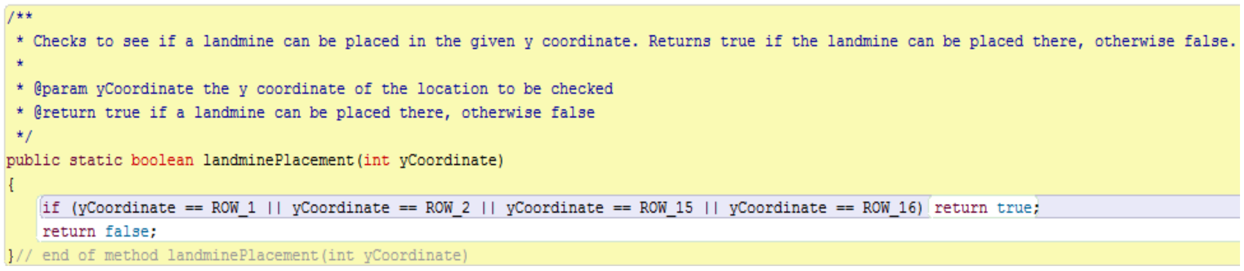
Determining whether a move or placement is legal

There are certain conditions in placing a piece, according to the rules. For a flag, the location must be a headquarter while for landmines, the location must be in the last 2 rows and the location must not already have a piece there. For normal pieces, the pieces cannot be placed on campsites and the location must not already have a piece there. For all placements, the piece must be placed on their side of the board.

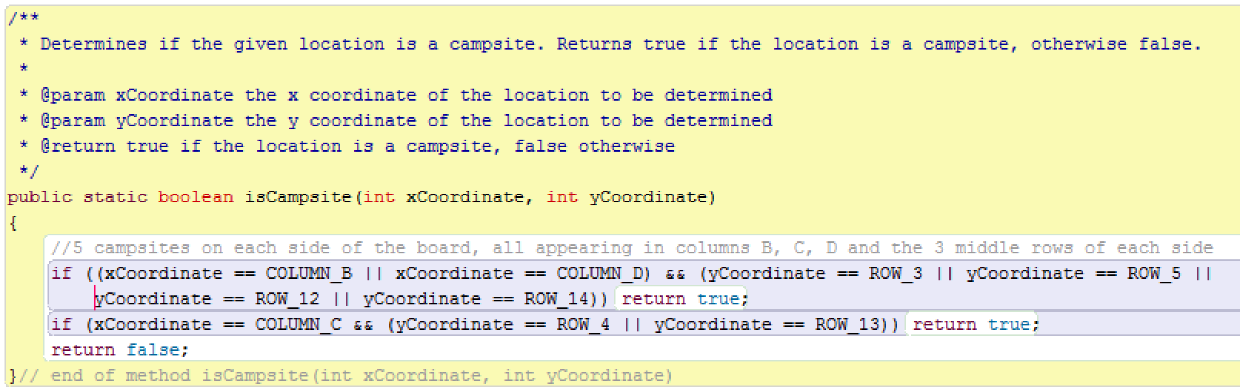
Therefore, methods were constructed to determine if a location was on the campsite, headquarters or in the last 2 rows.



```java
/**
 * Determines if the given location is a headquarter. Returns true if the location is a headquarter, otherwise false.
 *
 * @param xCoordinate the x coordinate of the location to be determined
 * @param yCoordinate the y coordinate of the location to be determined
 * @return true if the location is a headquarter, false otherwise
 */
public static boolean isHeadquarter(int xCoordinate, int yCoordinate)
{
    if ((xCoordinate == COLUMN_B || xCoordinate == COLUMN_D) && (yCoordinate == ROW_1 || yCoordinate == ROW_16)) return true;
    return false;
}// end of method isHeadquarter(int xCoordinate, int yCoordinate)
```



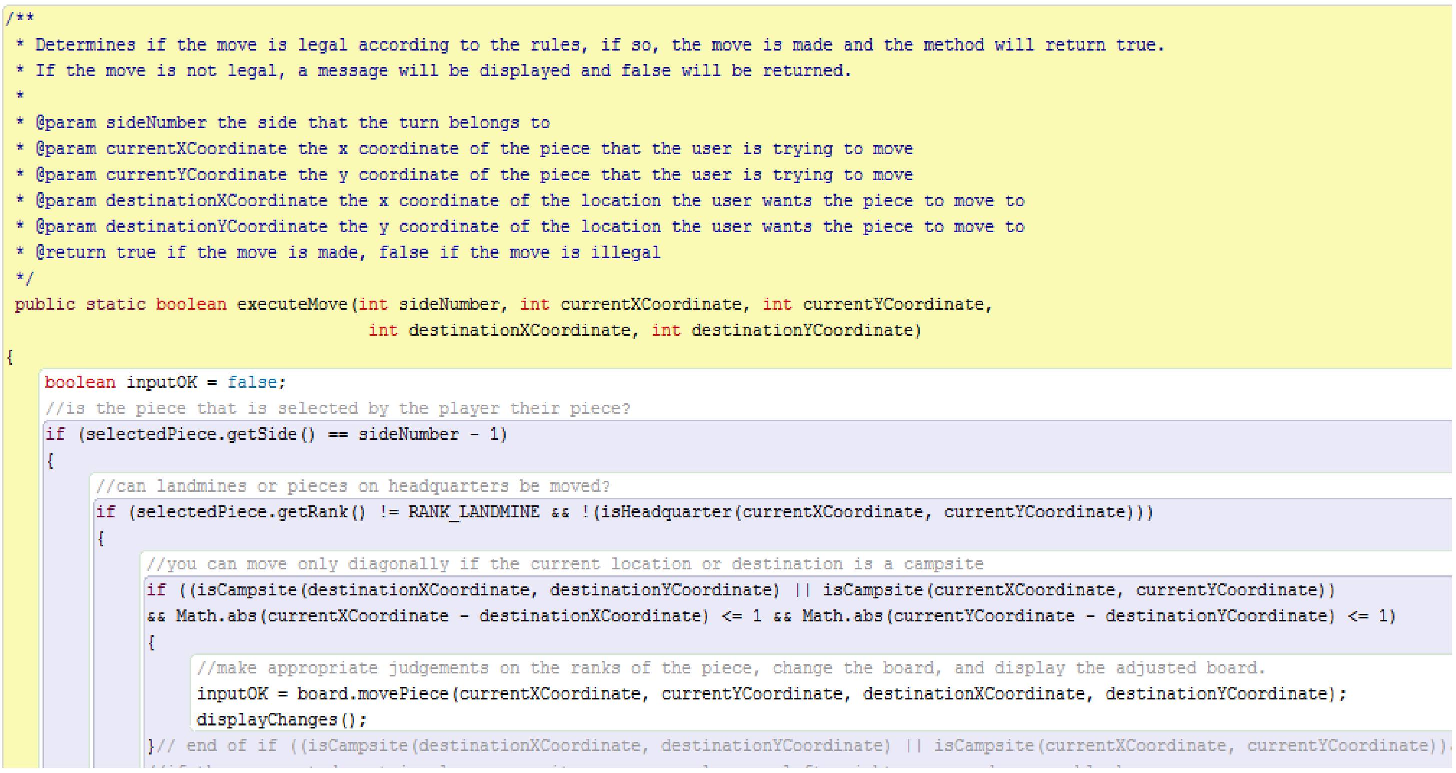
```java
/**
 * Checks to see if a landmine can be placed in the given y coordinate. Returns true if the landmine can be placed there, otherwise false.
 *
 * @param yCoordinate the y coordinate of the location to be checked
 * @return true if a landmine can be placed there, otherwise false
 */
public static boolean landminePlacement(int yCoordinate)
{
    if (yCoordinate == ROW_1 || yCoordinate == ROW_2 || yCoordinate == ROW_15 || yCoordinate == ROW_16) return true;
    return false;
}// end of method landminePlacement(int yCoordinate)
```



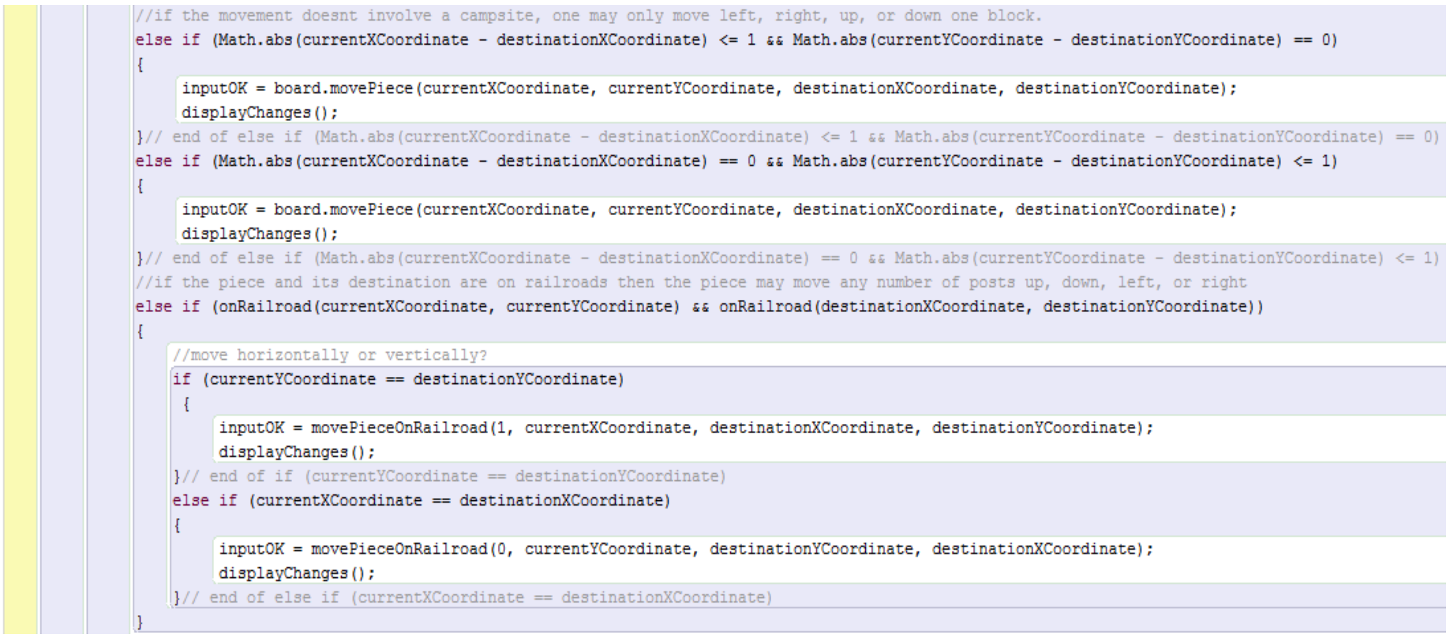
```java
/**
 * Determines if the given location is a campsite. Returns true if the location is a campsite, otherwise false.
 *
 * @param xCoordinate the x coordinate of the location to be determined
 * @param yCoordinate the y coordinate of the location to be determined
 * @return true if the location is a campsite, false otherwise
 */
public static boolean isCampsite(int xCoordinate, int yCoordinate)
{
    //5 campsites on each side of the board, all appearing in columns B, C, D and the 3 middle rows of each side
    if ((xCoordinate == COLUMN_B || xCoordinate == COLUMN_D) && (yCoordinate == ROW_3 || yCoordinate == ROW_5 ||
        yCoordinate == ROW_12 || yCoordinate == ROW_14)) return true;
    if (xCoordinate == COLUMN_C && (yCoordinate == ROW_4 || yCoordinate == ROW_13)) return true;
    return false;
}// end of method isCampsite(int xCoordinate, int yCoordinate)
```

For determining whether a turn is legal or not, the player's input is split into the x and y coordinates. These coordinates are passed on to a method containing a set of if statements, which determine whether the movement is legal, and what action the program will take to make the move if it is legal.

The method is on the next page.



```java
/**
 * Determines if the move is legal according to the rules, if so, the move is made and the method will return true.
 * If the move is not legal, a message will be displayed and false will be returned.
 *
 * @param sideNumber the side that the turn belongs to
 * @param currentXCoordinate the x coordinate of the piece that the user is trying to move
 * @param currentYCoordinate the y coordinate of the piece that the user is trying to move
 * @param destinationXCoordinate the x coordinate of the location the user wants the piece to move to
 * @param destinationYCoordinate the y coordinate of the location the user wants the piece to move to
 * @return true if the move is made, false if the move is illegal
 */
 public static boolean executeMove(int sideNumber, int currentXCoordinate, int currentYCoordinate,
                                   int destinationXCoordinate, int destinationYCoordinate)
{
    boolean inputOK = false;
    //is the piece that is selected by the player their piece?
    if (selectedPiece.getSide() == sideNumber - 1)
    {
        //can landmines or pieces on headquarters be moved?
        if (selectedPiece.getRank() != RANK_LANDMINE && !(isHeadquarter(currentXCoordinate, currentYCoordinate)))
        {
            //you can move only diagonally if the current location or destination is a campsite
            if ((isCampsite(destinationXCoordinate, destinationYCoordinate) || isCampsite(currentXCoordinate, currentYCoordinate))
            && Math.abs(currentXCoordinate - destinationXCoordinate) <= 1 && Math.abs(currentYCoordinate - destinationYCoordinate) <= 1)
            {
                //make appropriate judgements on the ranks of the piece, change the board, and display the adjusted board.
                inputOK = board.movePiece(currentXCoordinate, currentYCoordinate, destinationXCoordinate, destinationYCoordinate);
                displayChanges();
            }// end of if ((isCampsite(destinationXCoordinate, destinationYCoordinate) || isCampsite(currentXCoordinate, currentYCoordinate))
```



```java
            //if the movement doesnt involve a campsite, one may only move left, right, up, or down one block.
            else if (Math.abs(currentXCoordinate - destinationXCoordinate) <= 1 && Math.abs(currentYCoordinate - destinationYCoordinate) == 0)
            {
                inputOK = board.movePiece(currentXCoordinate, currentYCoordinate, destinationXCoordinate, destinationYCoordinate);
                displayChanges();
            }// end of else if (Math.abs(currentXCoordinate - destinationXCoordinate) <= 1 && Math.abs(currentYCoordinate - destinationYCoordinate) == 0)
            else if (Math.abs(currentXCoordinate - destinationXCoordinate) == 0 && Math.abs(currentYCoordinate - destinationYCoordinate) <= 1)
            {
                inputOK = board.movePiece(currentXCoordinate, currentYCoordinate, destinationXCoordinate, destinationYCoordinate);
                displayChanges();
            }// end of else if (Math.abs(currentXCoordinate - destinationXCoordinate) == 0 && Math.abs(currentYCoordinate - destinationYCoordinate) <= 1)
            //if the piece and its destination are on railroads then the piece may move any number of posts up, down, left, or right
            else if (onRailroad(currentXCoordinate, currentYCoordinate) && onRailroad(destinationXCoordinate, destinationYCoordinate))
            {
                //move horizontally or vertically?
                if (currentYCoordinate == destinationYCoordinate)
                 {
                    inputOK = movePieceOnRailroad(1, currentXCoordinate, destinationXCoordinate, destinationYCoordinate);
                    displayChanges();
                }// end of if (currentYCoordinate == destinationYCoordinate)
                else if (currentXCoordinate == destinationXCoordinate)
                {
                    inputOK = movePieceOnRailroad(0, currentYCoordinate, destinationYCoordinate, destinationXCoordinate);
                    displayChanges();
                }// end of else if (currentXCoordinate == destinationXCoordinate)
            }
```

Word Count: 891