· What needed to be improved? That is, what “bad code smell” was detected? Use the terminology
found in Chapter 3 of the Fowler text.

Large class: Original version of Matrix class had all methods at one class.

Duplicate code: Same functionalities were used multiple times in code not in a form of function.

Long method: Code had long method that it was hard to see what is going on.

Encapsulation: Make it hard to update the compoents

Comments that explain bad code: Name of code that mislead what does the function do.

· What refactoring was applied? What steps did you follow? Use the terminology and mechanics outlined in the Fowler text, and illustrate the process with well-chosen code fragments taken from particular versions of your system.

Large class: I extracted MatrixPrint, MatrixHandleInput classes from matrix class since all the functions were in one class.

Identify related functionalities in Matrix class first. Create class and move related function into that class. Then adjust class dependencies and interactions.

```
MatrixHandleInput handleInput;

        public Matrix() {
                this.handleInput = new MatrixHandleInput();
        }

        private MatrixPrint handlePrint = new MatrixPrint();
```

I have created class called MatrixPrint, MatrixHandleInput and matrixOperatorCalculation and move the functions to that classes so we clearly see what each class do in matrix.

Duplicate code: I extracted common functionalities like initializeMatrix that was used in each calculation in the function.

First, search for redundant code and then extract that code into a method. Then replace all redundant instance into that method.

```
matrix1 = new int[row1][col1];

        for (int i =0; i< row1;i++){

        for (int j=0; j<col1;j++){
```

```
                System.out.print("Enter Matrix 1 [" + (i+1) + "][ "+ (j+1) + "]:");

                matrix1[i][j]= checkInputInt();

            }

            }
```

it was initializing in every function which has redundant code. So I make it as method so it can be easily used without writing same code every single time.

```
public int[][] initializeMatrix(int rowCount, int colCount) {
        int[][] matrix = new int[rowCount][colCount];
        for (int i = 0; i < rowCount; i++) {
            for (int j = 0; j < colCount; j++) {
             System.out.print("Enter Matrix [" + (i + 1) + "][" + (j + 1) + "]: ");
             matrix[i][j] = checkInputInt();
            }
        }
        return matrix;
            }
```

Long Method: I break the long method into smaller form which improved readability. The length of performMatrixForAddition function code has been reduced.

```
private void getMatrixForAddition(){

        System.out.println("-------------Addition------------");

                System.out.print("Enter row matrix 1:");

                row1 = checkArrayElements();

                System.out.print("Enter column matrix 1:");

                col1 = checkArrayElements();

                matrix1 = new int[row1][col1];

                for (int i =0; i< row1;i++){

                for (int j=0; j<col1;j++){

                System.out.print("Enter Matrix 1 [" + (i+1) + "][ "+ (j+1) + "]:");

                matrix1[i][j]= checkInputInt();

                }

                }

                boolean isRowValid=false;

                boolean isColValid=false;
```

```
while (!isRowValid){

System.out.print("Enter row matrix 2:");

row2=checkArrayElements();

if (row1==row2) isRowValid=true;

else System.out.println("Invalid Matrix for addition, both matrix must have the same number of row.");

}

while (!isColValid){

System.out.print("Enter column matrix 2:");

col2=checkArrayElements();

      if (col2==col1) isColValid=true;

else System.out.println("Invalid Matrix for addition, both matrix must have the same number of column.");

}

matrix2=new int[row2][col2];

for (int i =0; i< row2;i++){

for (int j=0; j<col2;j++){

System.out.print("Enter Matrix 2 [" + (i+1) + "][ "+ (j+1) + "]:");

matrix2[i][j]= checkInputInt();

}

}

addMatrix(matrix1, matrix2, row1, col1);

}
```

This was the code for getMatrixForAddition()

```
private void performMatrixForAddition(){
    System.out.println("-------------Addition------------");
    setRow1(getMatrixDimension("Enter row matrix 1: "));
    setCol1(getMatrixDimension("Enter column matrix 1: "));
    setMatrix1(handleInput.initializeMatrix(getRow1(), getCol1())); //Extract Method
    handleInput.validateMatrixForAddOrSub(getRow1(), getCol1());
    setMatrix2(handleInput.initializeMatrix(getRow2(), getCol2())); //Extract Method
    addMatrix(getMatrix1(), getMatrix2(), getRow1(), getCol1());
        }
```

I extract method for each line so I can reduce the number of line.

Rename: Some of the names are not descriptive that may not realize what does this function do by name.

Identify misleading names and I change misleading names to more intuitive name.

Encapsulation: I created getters and setters so it increased flexibility and control also reduce complexity to debug.

· How was the code tested?

Code was tested by creating test of function whether it works properly like it worked as previously.

· Why is the code better structur First, search for redundant code and then extract that code into a method. Then replace all redundant instance into that method.

ed after the refactoring? Does the result of the refactoring suggest or enable further refactorings?

The new structure of this code improves the modularity. Each class and methods are separated that know what each class and method do in code.

It increase the maintainability since code is more clear to see the structure which make it better for future modification.

Now when people sees the code, it improved readability that it reduced lines of each method that it is more easier to understand.