Evaluate the test plan 5061:

In this report, I will evaluate a test plan by testing it on another code.

The test plan I used, has the number 5061, and the code I ran the test plan for it, has the number 9455.

At first, I am going to evaluate the test code, then I will run the code in a different scenario and check the functionality of the test plan. I will provide a result for this test and show how efficient the test is and how robust the code was written.

In this test plan all the methods read the inputs from a text file. There is no parsing method in the code. So, the inputs must be created for this test. All the values in the file must be separated with space. ',' or other delimiters are not accepted.

Section 1:

In this section, the test plan tries to validate minimum number of input integers needed to create a table. The input can be a 2 x 2 format (row and column) and the expected output will be Fail.

Section 2:

In this section, the test plan tries to validate number of input integers that will not create a table. JUnit is needed for testing. The input file will be a file that doesn't have any square number of the elements. Expected output here will be Fail.

Section 3:

In this section, test plan tries to Validate creating a table with four positive integers as input.

The input file can be any array with the integer values and square number of the elements. The expected elements will be True.

Section 4:

In this section, test plan checks output of the code when we give it an array with the negative values. This test will validate creating a table with four negative integers as input. In this test case, the test plan tries to identify the correctness of the code by handling the error. So, it use **assertEquals(true,true)** statement but, the handling the test in the catch block.

Section 5:

In this case. The test plan tries to check a mixed array as an input. This input can be zero or any other integer numbers. So, test will validate creating a table with four mixed sign integers and a zero as input.

In this case, the test tries to check the accuracy of the function by handling the error. So, if there is any unexpected output, the try and catch mechanism can catch it.

Section 6:

In this section, test plan checks the input if it is sorted or not. This test will validate if a set of input values are in order sequence. Expected value here is true if the input is sorted. So, it compares the result of the isSorted function with true value here.

Section 7:

In this section, the test plan checks functionality of the code in deal with an unsorted array. The expected result is False, when we check the isSorted function with an un-sorted input. This test will validate if a set of input values are not in order sequence. Expected output is false, due to set of input numbers not sorted.

Analyzing the test code:

The test is written on Java. It imports the JUnit and uses this package to automate the testing. Using JUnit gives the programmer the advantage of checking the code during writing the code and run it simultaneously to be sure of the accuracy and functionality of the code.

The programmer imports an un-used class of JUnit (org.junit.Before). It would be better to remove the unused package to prevent any unnecessary load on the program.

In the code, the programmer calls some files (apparently the text file of the table's values) but, unfortunately, they cannot be found in the submitted file. So, we add this file manually to able run the code.

Test Case 1:

Checking if the function **isSorted** can correctly determine a sorted array or not

> Input: 6 7 9 13 21 45 101 102 150 (a sorted array with square dimension)
>
> Result: it correctly returns a true value
>
> Function: In this test assertEquals uses to check the equality between a given sorted array and true.
>
> assertEquals(true, TableSorter.isSorted(Table.GetTable("sorted_array.txt")));

Test Case 2:

Checking if the function **isSorted** can correctly determine a not sorted array or not.

> Input: 13 7 6 45 21 9 101 102 67 (a not sorted array with square dimension)
>
> Result: it correctly returns the false value
>
> Function: In this test assertEquals uses to check the result of **isSorted** with false.
>
> assertEquals(false, TableSorter.isSorted(Table.GetTable("unsorted_array.txt")));

Test Case 3:

Checking the function's behavior to the non-Int array.

Input: 6.2 7.1 9.5 -13 21.1 45.01 101.0 102.1 150.01 (an array with double digits)

Result: It gets an exception, but the test cannot handle it and cause an error in the test. Fail result. By catching the error, this problem can be solved.

Function: this test uses the assertEquals and compares the return value with false.

assertEquals(false, TableSorter.isSorted(Table.GetTable("nonInt_array.txt")));

Test Case 4:

Checking if the program is robust when the input is a negative array or not.

Input: -602 -107 -92 -23 -21 -15 -11 -10 –5 (negative integer array).

Result: It returns false. That means the program cannot handle negative integer and the test catch this error correctly.

Function: this test uses assertEqualitys and compares the result by false value.

assertEquals(false, TableSorter.isSorted(Table.GetTable("negativeInt_array.txt")));

Test Case 5:

In this test case, the program checks about the positive Integer array.

Input: 6 7 9 13 21 45 101 102 150 (a sorted positive integer array)

Result: It returns true, but the function compares the result with false. This is wrong and it must compare with true. So, the test case fails in this case too.

Test Case 6:

In this test case, the test checks the behavior of the program when the inputs are not in a square number.

Input: 6 7 9 13 21 45 101 102 (not in a square number of elements)

Result: the test throws an error. It would be better to catch the error. But the program can find an array without the square number of the inputs.

Test Case 7:

In this test case, the program checks an input with the square number of the elements.

Input: 6 7 9 13 21 45 101 102 150

Result: the test correctly returns the true value. This means the function is worked with the square number of elements as an input.

Test Case 8:

The objective of this test case is to check function behavior with one element as an input.

Input: 4

Result: It returns true correctly and that means the function can handle one single element as an input too

Conclusion:

The test works well on all 8 different test cases. In two test cases (3, 6) it would be better to handle the exception in the test function does not throw it out into the console. In test case 5, the value of false must change to true.

The rest of the code works well, and it can check the functionality of TableSorter very well.

The test plan documented very well and clear for the user.