TEAM:

Bledea Mihaela Alexandra

Bretan Cezar Alexandru

Rusu Ilie Alexandru

**Black Box Testing – findSadFeeling method**

We first started with a black box thought process where we disregarded, temporarily, the code to think out the requirements of the function. This enabled us to develop testing scenarios that

were not initially treated in the code. For example, we haven’t considered the scenario where the index received by the method might be negative. This discovery protects further users from ArrayIndexOutOfBounds Exception.

**Equivalence classes:**

For the **findSadFeeling(int index)** method, the input domain can be partitioned into several equivalence classes based on the method's behavior for different index values. The method expects an integer index as input and performs a search operation starting from the specified index. Here are the potential equivalence classes we discovered:

1. Negative values: This represents an invalid class. Negative index values would likely throw an IndexOutOfBoundsException, as indices for lists in Java are 0-based.
2. Zero to emotions.size() - 1: This represents a valid class. The method should start the search from the specified index and either return the index of the first -1 value or emotions.size() if no -1 is found.
3. Values equal to or greater than emotions.size(): This is another invalid class. This case would normally throw an IndexOutOfBoundsException if not treated correctly, but we solved this problem by returning the size of this array directly in such a case.

**Boundary Values Analysis**

Here, in the findSadFeeling(int index) method, we have one input variable: index, which is the starting index for searching the emotions list for a -1 (representing a "sad feeling"). The acceptable range for the index is from 0 to emotions.size() - 1. The possible BVAs that we found are:

Let’s suppose we have an array with 5 elements:

* Lower boundary index value: 0
* Boundary index value within 2
* Upper boundary index value: 4
* Exceeding lower boundary index value: -1
* Exceeding upper boundary index value: 5

Now, when was time to properly design the test cases, we made a comparison between the equivalence classes found and boundary values cases found and we saw that the cases for BV were all included in the equivalence classes cases found, so we decided to implement the tests for the BV because they had somehow an extended vision over what could go wrong and we would be more sure we cover all the possible points of failure. The test cases we design are:

1. **Lower boundary index value - testFindSadFeelingFirstInList()**

**Input:** 0 (lower boundary), size of the array = 4, array = [-1, 0, 0, 1]

**Expected Result:** Returns the index of the first -1 found starting from index 0, which is 0.

**Reason:** This test case checks the method's behavior at the lower boundary of the input domain.

1. **Upper boundary index value - testFindSadFeelingLastInList()**

**Input**: 3 (upper boundary), size of the array = 4, array = [1, 0, 0, -1]

**Expected Result**: Returns 3 because the value at this index is -1.

**Reason:** This test case checks the method's behavior at the upper boundary of the input domain.

1. **Boundary index value within - testFindSadFeelingWithinBV()**

**Input:** 2 (Boundary index value within), size of the array = 5, array = [0, 0, -1, 1, 0]

**Expected Result:** Returns the index of the first -1 found starting from index 1, which is index = 2.

**Reason:** This test case checks the method's behavior within the boundary values.

1. **Exceeding lower boundary index value - testFindSadFeelingLowerThanBV()**

**Input**: -1, size of the array = 5, array = [0, 0, -1, 1, 0]

**Expected Result:** Returns emotions.size(), which is 5

**Reason**: This test case checks the method’s behavior at the exceeding lower boundary of the input domain

1. **Exceeding upper boundary index value - testFindSadFeelingHigherThanBV()**

**Input** = 7, size of the array = 5, array = [0, 0, -1, 1, 0]

**Expected Result:** Returns emotions.size(), which is 5

**Reason:** This test case checks the method’s behavior at the exceeding upper boundary of the input domain

**White Box Testing – checkNeighbours method**

After the first phase, we started considering code. As such, we explored all the possible branches to take out cases that we might have missed in the first scenario. Specific bugs were not discovered during this phase, but we could effectively treat case scenarios according to the implementation. To support white-box testing, we have noted every statement, every condition, and every decision for the function’s code as shown in the figure below:

A picture containing text, screenshot, purple, violet

Description automatically generated

Further, we’ve created the flow graph that can be seen above:

**A picture containing diagram, line, circle

Description automatically generatedA picture containing text, screenshot, font, number

Description automatically generated**

After creating the flow graph we were able to find all the individual paths, and below we’ve attached some images for the test cases we’ve made, along with the condition coverage, decision coverage, path coverage, and loop coverage.

**A picture containing text, font, number, line

Description automatically generated**

**A picture containing text, screenshot, colorfulness, line

Description automatically generated**

**Integration testing**

Bottom-up integration testing is a strategy used to test integrated modules in a system. The testing process starts from the lower-level components and gradually moves towards higher-level components. With this approach, individual modules or units of the system are tested and validated independently before being integrated into larger subsystems or the overall system.

Bottom-up integration testing would be beneficial for testing the beHappy() method because it follows a modular and incremental approach to testing.

In the given code, the Service class depends on the Repository class, as it utilizes the methods from the Repository class to perform its operations. The beHappy() method, in particular, relies on the functionality provided by the findSadFeeling(), checkNeighbours(), and surroundByHappyFeeling() methods of the Repository class.

By using the bottom-up integration testing approach, the individual components (methods) of the Repository class can be thoroughly tested in isolation first. This ensures that each method works correctly on its own and produces the expected results. Any potential bugs or issues specific to the Repository class can be identified and fixed during this phase.

Once the components of the Repository class have been validated, the integration testing process can proceed to test the interactions and integration between the Service and Repository classes. Since the functionality of the beHappy() method heavily depends on the correct functioning of the Repository class, ensuring the integrity of this integration is crucial.

Bottom-up integration testing allows for a systematic and controlled approach to testing, starting from the foundational components and gradually building up to higher-level functionality. It helps identify and address any integration issues or dependencies between the modules early on, enabling the detection of potential bugs or inconsistencies that may impact the beHappy() method.

Overall, bottom-up integration testing provides a comprehensive way to validate the individual components and their interactions within the system, making it a suitable approach to ensure the correctness and robustness of the beHappy() method in this scenario.