Software Quality and Testing - Lab 2

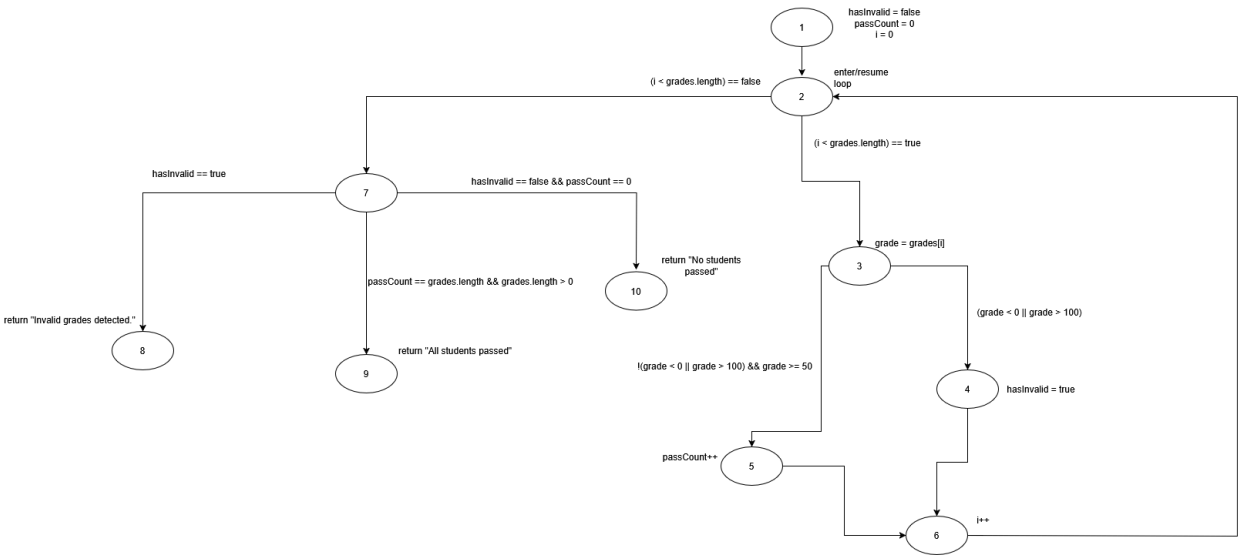
Марко Миновски - 221552

For the given function:

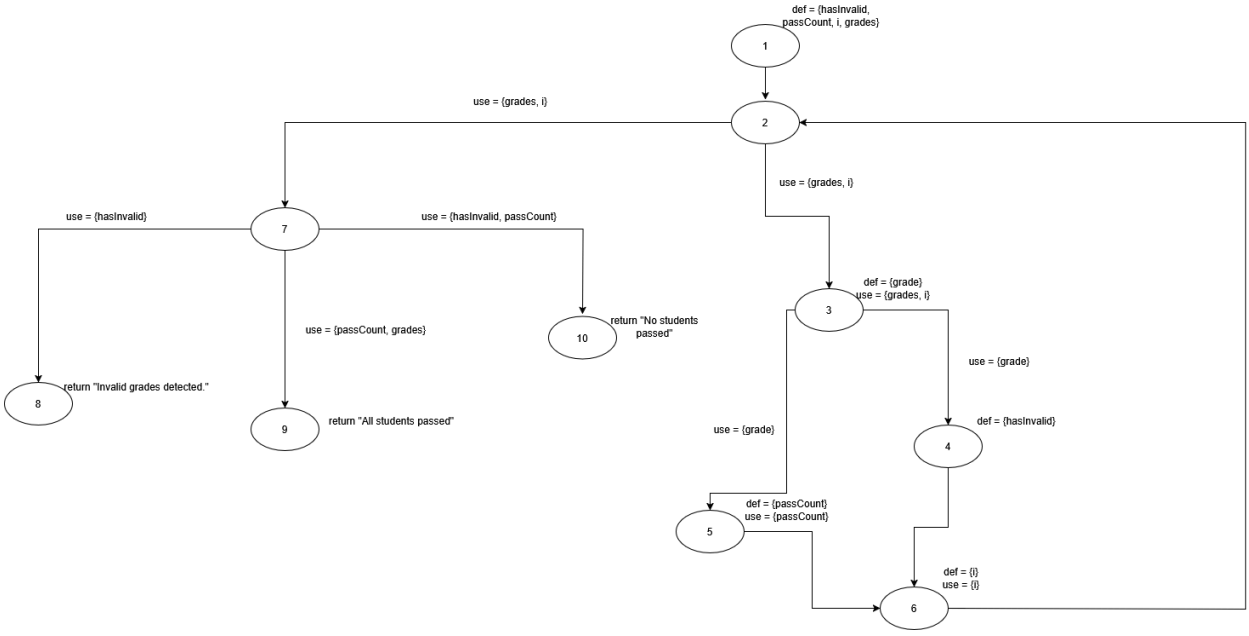
```
/**
 * Analyzes an array of student grades. Returns a summary message based on: - Number of passing grades (>= 50) - If any grade is invalid (< 0 or > 100) - If all students passed
 *
 * Params: grades – an array of integers representing student grades
 *
 * Returns: summary message
 */
public static String analyzeGrades(int[] grades) {
    boolean hasInvalid = false;
    int passCount = 0;

    for (int i = 0; i < grades.length; i++) {
        int grade = grades[i];

        if (grade < 0 || grade > 100) {
            hasInvalid = true;
        } else if (grade >= 50) {
            passCount++;
        }
    }

    if (hasInvalid) {
        return "Invalid grades detected.";
    } else if (passCount == grades.length && grades.length > 0) {
        return "All students passed.";
    } else if (passCount == 0) {
        return "No students passed.";
    } else {
        return "Some students passed.";
    }
}
```

1. Graph

- “Regular”



- Data flow coverage



2. find all du-paths that satisfy the criteria for All-Du-Paths Coverage

- I will follow the procedure in auditory exercises and write in two separate tables def and use instances at each node, and then at each edge respectively.

Node	def	use
1	{hasInvalid, passCount, i, grades}	/
3	{grade}	{grades, i}
4	{hasInvalid}	/
5	{passCount}	{passCount}
6	{i}	{i}

Edge	def	use
(2,3)	/	{grades, i}
(2,7)	/	{grades, i}
(3,4)	/	{grade}
(3,5)	/	{grade}
(7,8)	/	{hasInvalid}
(7,9)	/	{passCount, grades}
(7,16)	/	{hasInvalid, passCount}

- We assume that if a node or edge does not appear at their respective table, then there neither a definition nor usage of any variables at that node/edge

Du-pairs

Variable	Du-pairs
hasInvalid	[1, (7,8)] [1, (7,9)] [4, (7,8)] [4, (7,9)]
passCount	[1, (7,8)] [1, (7,10)] [5,5] [5, (7,8)] [5, (7,10)]
grades	[1, (2,3)] [1, (2,7)] [1, 3]
grade	[3, (3,4)] [3, (3,5)]
i	[1, (2,3)] [1, (2,7)] [1, 3] [6, 6] [6, (2,3)] [6, (2,7)] [6, (2,3)] [6, 3]

After filtering, we get the following du-paths:

```
1: [1, 2, 3, 4, 6, 2, 7, 8]
2: [1, 2, 3, 5, 6, 2, 7, 9]
3: [1, 2, 3, 5, 6, 2, 7, 10]
4: [5, 6, 2, 3, 5]
5: [6, 2, 3, 5, 6]
6: [6, 2, 3, 4, 6]
7: [3, 4, 6, 2, 3]
```

The 1st, 2nd and 3rd du-paths are also test-paths, so we use those for all-du-path coverage.

For 4, we use the following test path (This also covers 5):

- `[1, 2, 3, 5, 6, 2, 3, 5, 6, 2, 7, 9]`

For 6, we use the following test path (This also covers 7):

- `[1, 2, 3, 4, 6, 2, 3, 4, 6, 7, 8]`

In total:

```
1: [1, 2, 3, 4, 6, 2, 7, 8]
2: [1, 2, 3, 5, 6, 2, 7, 9]
3: [1, 2, 3, 5, 6, 2, 7, 10]
4: [1, 2, 3, 5, 6, 2, 3, 5, 6, 2, 7, 9]
5: [1, 2, 3, 4, 6, 2, 3, 4, 6, 7, 8]
```

With these 5 tests, we achieve all-du-path coverage.

3. find the minimal test set that achieves Prime Path Coverage and create real Junit tests

Simple paths

[1, 2, 7, 10]
[1, 2, 3, 5, 6]
[1, 2, 3, 4, 6]
[3, 4, 6, 2, 7, 8]
[3, 4, 6, 2, 7, 9]
[3, 4, 6, 2, 7, 10]
[2, 3, 4, 6, 2]
[2, 3, 5, 6, 2]
[6, 2, 3, 4, 6]
[6, 2, 3, 5, 6]
[5, 6, 2, 3, 5]
[1, 2, 3, 5]
[1, 2, 3, 4]
[1, 2, 7, 8]
[1, 2, 7, 9]
[3, 4, 6]
[3, 4, 6, 2, 3]
[3, 5, 6, 2, 3]
[5, 6, 2]
[5, 6, 2, 7, 8]
[5, 6, 2, 7, 9]
[5, 6, 2, 7, 10]

[6, 2, 7, 8]
[6, 2, 7, 9]
[6, 2, 7, 10]
[5, 6, 2, 7]
[4, 6, 2, 3, 4]
[4, 6, 2, 7, 8]
[4, 6, 2, 7, 9]
[4, 6, 2, 7, 10]
[4, 6, 2, 3]
[4, 6, 2]
[6, 2, 3]
[6, 2, 7]
[2, 7, 8]
[2, 7, 9]
[2, 7, 10]
[1, 2, 7]
[7, 8]
[7, 9]
[7, 10]
[4, 6]
[3, 4]
[1, 2]
[2, 3]
[3, 5]
[5, 6]
[6, 2]
[1]

[2]
[3]
[4]
[5]
[6]
[7]
[8]
[9]
[10]

‘

Prime paths

[2, 3, 4, 6, 2]
[2, 3, 5, 6, 2]
[6, 2, 3, 4, 6]
[6, 2, 3, 5, 6]
[5, 6, 2, 3, 5]
[3, 5, 6, 2, 7, 9]
[3, 5, 6, 2, 7, 8]
[3, 5, 6, 2, 7, 10]
[3, 4, 6, 2, 7, 8]
[3, 4, 6, 2, 7, 9]
[3, 4, 6, 2, 7, 10]
[3, 4, 6, 2, 3]
[3, 5, 6, 2, 3]
[4, 6, 2, 3, 4]
[6, 2, 3, 4, 6]
[6, 2, 3, 5, 6]
[5, 6, 2, 3, 4]
[4, 6, 2, 3, 5]
[1, 2, 7, 8] *(Impossible)*
[1, 2, 7, 9] *(Impossible)*
[1, 2, 7, 10]

Minimal test set to satisfy:

Some of the above prime paths are impossible to execute as test paths. For example, it is impossible, in our function, for the end result to be “Invalid grades detected” without any iterations of the loop (path: `[1, 2, 7, 8]`). After filtering out impossible paths, we are left with:

- Test 1: path `[1, 2, 3, 4, 6, 2, 3, 5, 6, 2, 7, 8]`
- Test 2: path `[1, 2, 3, 5, 6, 2, 3, 4, 6, 2, 7, 8]`
- Test 3: path `[1, 2, 3, 5, 6, 2, 3, 5, 6, 2, 7, 9]`
- Test 4: path `[1, 2, 3, 5, 6, 2, 3, 5, 6, 2, 7, 10]`
- Test 5: path `[1, 2, 7, 10]`

We can also extend tests 1 through 4 to loop more than once, but it is unnecessary - the end result is the same. Our prime paths are covered, and the output is identical as well.

JUnit tests:

```
import static org.junit.jupiter.api.Assertions.assertEquals;

import com.skit.Main;
import org.junit.jupiter.api.Test;

public class MainTest {

    /**
     * Test1 - follows the test path: [1, 2, 3, 4, 6, 2, 3, 5, 6, 2, 7, 8] <br>
     * Expected input: Non-null array of size 2; First element invalid, second
valid <br>
     * Expected output: "Invalid grades detected." <br>
     */
    @Test
    void test1() {
        int[] grades = {-5, 60};
        String result = Main.analyzeGrades(grades);
        assertEquals("Invalid grades detected.", result);
    }

    /**
```

```
    * Test2 - follows the test path: [1, 2, 3, 5, 6, 2, 3, 4, 6, 2, 7, 8] <br>
    * Expected input: Non-null array of size 2; First element valid, second
invalid <br>
    * Expected output: "Invalid grades detected." <br>
    */
   @Test
   void test2() {
       int[] grades = {60, 105};
       String result = Main.analyzeGrades(grades);
       assertEquals("Invalid grades detected.", result);
   }
   /**
    * Test3 - follows the test path: [1, 2, 3, 5, 6, 2, 3, 5, 6, 2, 7, 9] <br>
    * Expected input: Non-null array of size 2; Both valid - both passing <br>
    * Expected output: "All students passed." <br>
    */
   @Test
   void test3() {
       int[] grades = {75, 80};
       String result = Main.analyzeGrades(grades);
       assertEquals("All students passed.", result);
   }

   /**
    * Test4 - follows the test path: [1, 2, 3, 5, 6, 2, 3, 5, 6, 2, 7, 10]
<br>
    * Expected input: Non-null array of size 2; Both valid - both failing <br>
    * Expected output: "No students passed." <br>
    */
   @Test
   void test4() {
       int[] grades = {40, 30};
       String result = Main.analyzeGrades(grades);
       assertEquals("No students passed.", result);
   }

   /**
    * Test5 - follows the test path: [1, 2, 7, 10] <br>
    * Expected input: Non-null array, but empty. <br>
    * Expected output: "No students passed." <br>
    */
   @Test
   void test5() {
       int[] grades = {};
       String result = Main.analyzeGrades(grades);
       assertEquals("No students passed.", result);
   }
}
```

- Made by: Marko Minovski
- Source code: https://github.com/MarkoMinovski/skit-lab2