Lab Report 04

Assignment 04 - Testing

Authors: Dennis Loska, Tony Dorfmeister, Ai Dong 05.12.2017

Part 1: Black and White Box Tests

1. Getting started:

In this exercise we have tested edge cases and normal cases for the class `Absolute.java`.

The edge cases are:

- `testZero()`
- `testMaxValue()`
- `testMinValue()`

The normal cases are:

- `testSignedValueOf1()`
- `testSignedValueOf2()`
- `testUnSignedValueOf1()`
- `testUnSignedValueOf2()`

From these test cases we can see that only 1 test is failing which is `testSignedValueOf1()`. This is because the conditional in `Absolute.absoluteValueOf(int x)` is not correct. We compare with `<` instead we should compare with `<=`. Also the value to be compared to during the condition should be 0 rather than -1. This way this test would be succesful. For that reason, the return value of -1 should be 1 and not -1 as well.

2. Black-box-test:

In order to test `GradingScale.class` we need to cd into the directory this file is saved to and run the application by typing `java GradingScale` into the Terminal. Then the application allows us to insert numerical values into the terminal line by line. This should print the corresponding Grades to the terminal as long as values between 0 and 100 are entered. At last the program should return the average when the value -1 is entered.

For each grade 2 values are tested, the minimum to receive the grade, the maximum to receive the grade. The avg of the received grades also needs to be tested.

So the equivalence classes for this application would be:

for normal cases:

- input values in range(100)
- input value -1 prints average

for illegal cases:

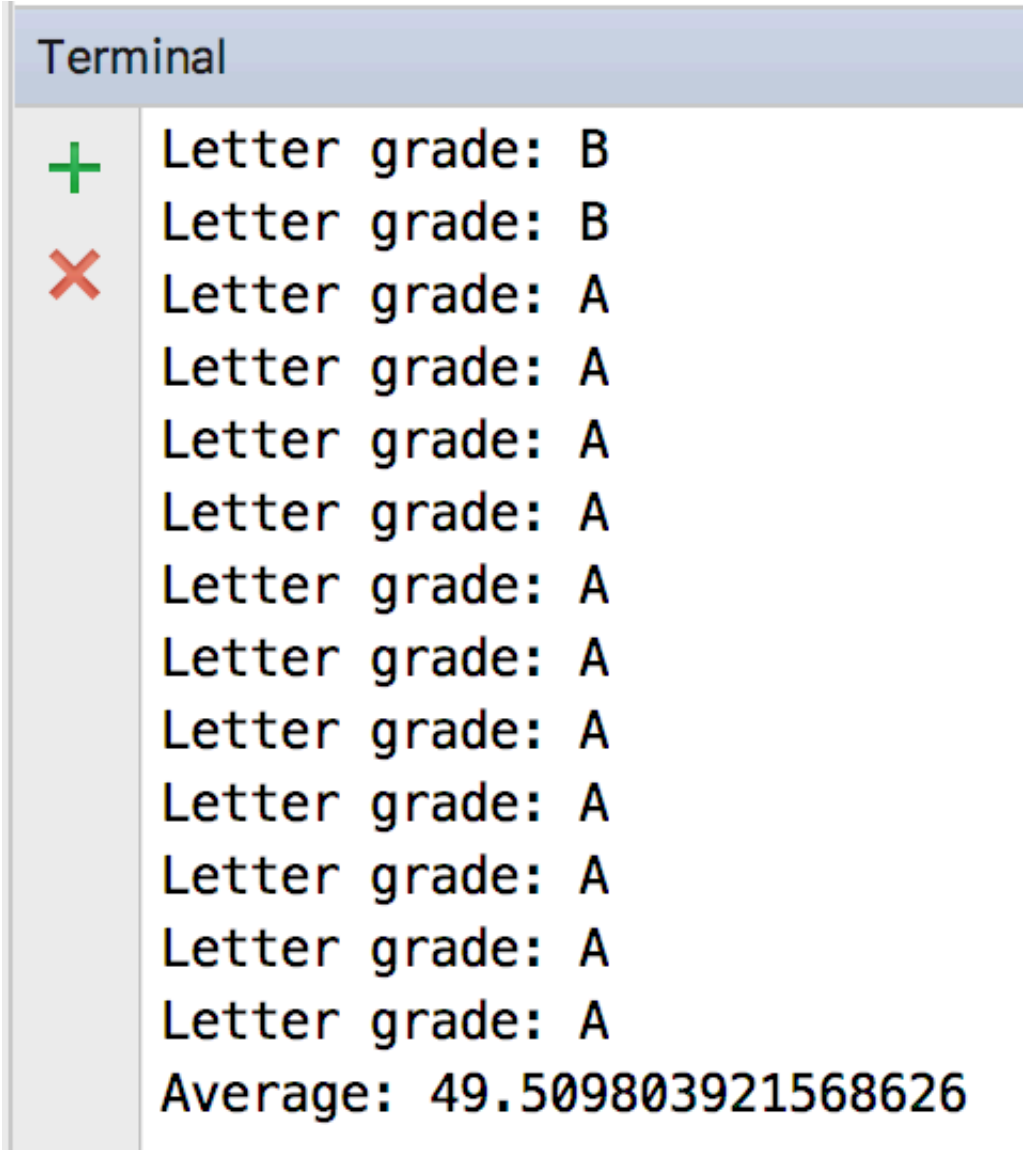
- input values <-1 and >100

Since the application is able to get multiple inputs at a time, we went ahead and populated generic textfiles with sample data that we can then feed into the GradingScale by using the pipe command `|` in bash.

The ouput is however off - after black box testing, we found the following issues:

50: F (instead of E) over 100: B (instead of error message) -1: AVG is calculated based on number of inputs overall instead of number of inputs related to grades.

Here are the results for the different tests:
test 01: input values in range 100



test 02: input values in range 100 without average

```
Terminal
Letter grade: A
Letter grade: A
Letter grade: A
Letter grade: A
Letter grade: A
Letter grade: A
Letter grade: A
Letter grade: A
Letter grade: A
Exception in thread "main" java.lang.NullPointerException
        at sun.misc.FloatingDecimal.readJavaFormatString(FloatingDecimal.java:1838)
        at sun.misc.FloatingDecimal.parseDouble(FloatingDecimal.java:110)
        at java.lang.Double.parseDouble(Double.java:538)
        at GradingScale.main(GradingScale.java:57)
```

This is the same output as test01 but now the loss of the value -1 at the end of file causes the program to return a NullPointerException

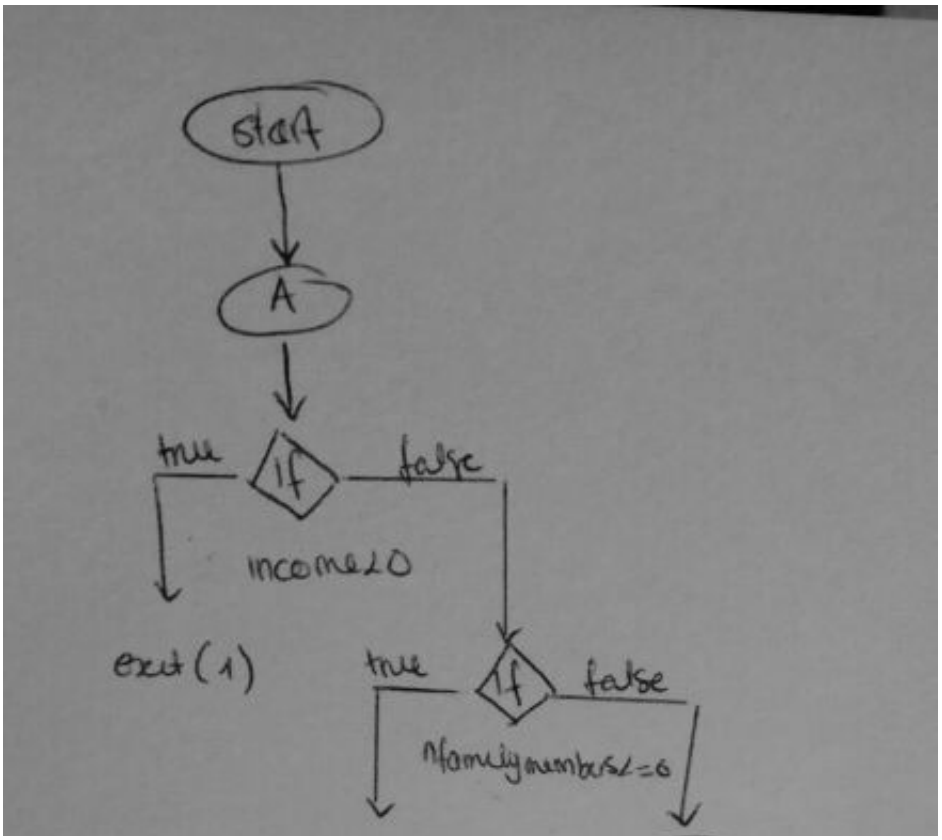
test 03: normal input values with -1 in the middle

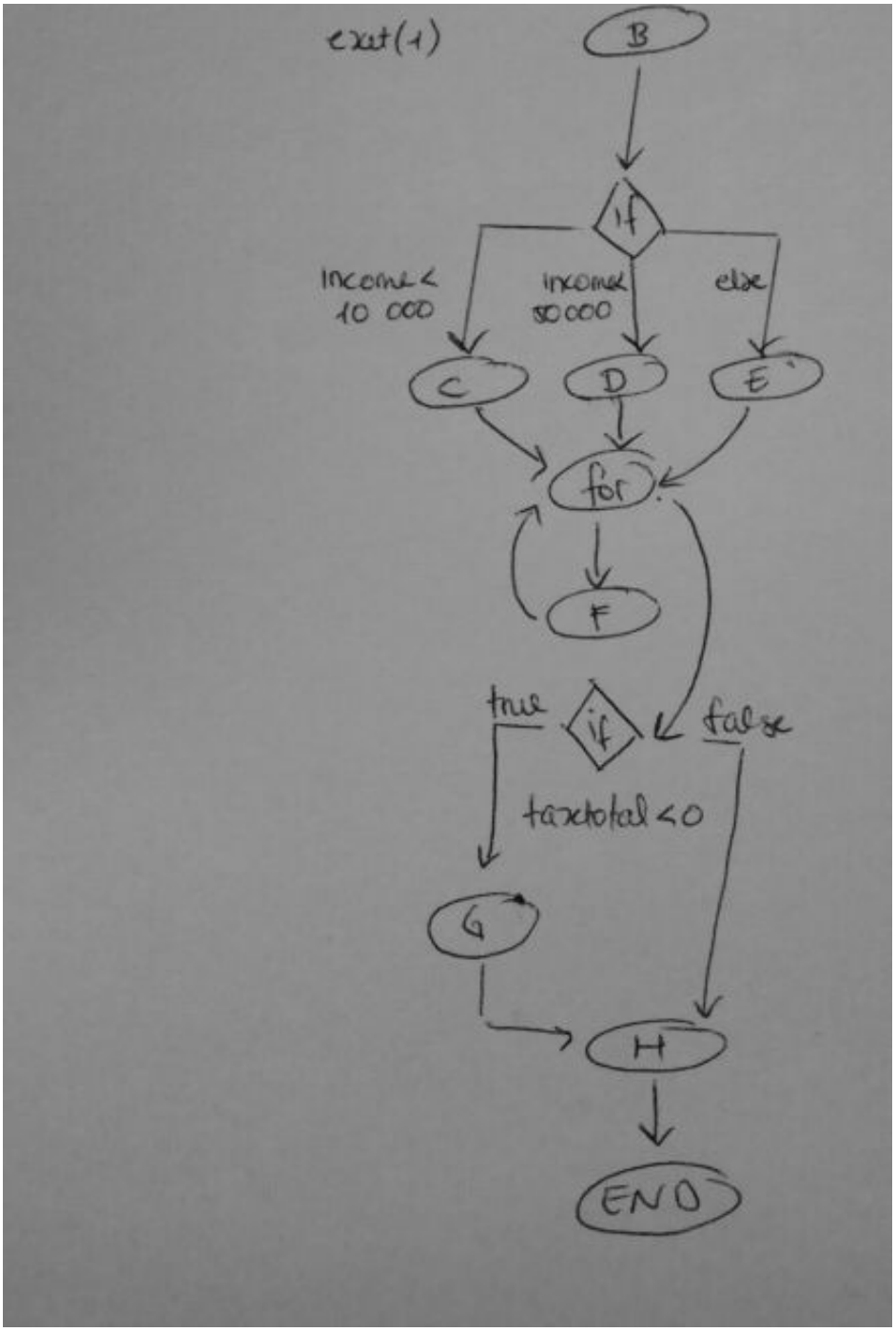
```
| ~/CloudStation/IMI/03_Semester/Informatik-03/labs/Lab04_Testing/a_black_and_white_box_tests @ Tonys-MacBook-Pro (tweak)
| => cat inputValuesWithAverageInMid.txt | java GradingScale
Enter your numeric grades as percentages one per line  and end with a negative number:
Letter grade: F
Letter grade: F
Letter grade: F
Average: 0.75
```

Here we can see that the program only runs

3. White-box-test:

In general, we can speak of up to 5 independent paths - as due to various contraints introducing new nodes. For clarification, please take a look at the code graph.






```
Overview of conditions
```

- if (income < 0)
  -if (nFamilyMembers <= 0)
  -if (income < 10000)
  else if (income < 50000)
  else
  -if (taxTotal < 0)

Test cases are actually a combination of the follow:

- (I): incomeLessThanZero()
- (III): FamilyMembersGreaterThanZero()
- (IV): FamilyMembersLessThanZero()

- (V): incomeIsZero()
- (VI): incomeLess10000ButGreaterZero()
- (VII): incomeLess50000ButGreater10000()
- (IX): taxTotalLessZero()
- (X): taxTotalGreaterZero()

Test Cases:

- (1) (I) -> exit(1)
- (2) -> (IV) -> exit(1)
- (3) -> (III) -> (V) -> (IX) -> end
- (4) -> (III) -> (VI) -> (X) -> end
- (5) -> (III) -> (VII) -> (X) -> end

4. Reflection:

Due to its progressive nature, one can hardly test inputs and outputs effectively. To test more effectively, the code would need to be refactored in a way, that each phase is represented by at least one method. This way, each phase could be tested independently.

Part 2: Test Driven Development

Deleting the first link

In order to delete a link, we first need to have a LinkedList filled with some nodes. So the test-case needs to create a list with nodes and then makes an attempt to delete the first one. This can be monitored using the *size* of the LinkedList

```

```

```
public void removeFromLinkedList(){
    LinkedList<String> ll = new LinkedList<>();
    ll.add(new Node<>("HUHU"));
    ll.add(new Node<>("HAHA"));
    ll.add(new Node<>("HIHI"));
    ll.remove();
    assertEquals(2,ll.getListLength());
}
//...the actual method:
void remove(){
    //deletes the first link
  if (head != null){
      head = head.next;
      listLength--;
  }
}
```

```
```

Reversing a List

As the assignment told us we started by implementing test cases for a linked list. For this we created a new class called `LinkedListTest`. In there we created tests for reversing a single item list, an empty list and a multiple item list. Because we chose to implement a generic LinkedList we also created a test for reversing a mixed data type list. While implementing the method `reverse()` we have noticed that the LinkedList was lacking an `add()` method so we went ahead created a test and implemented that method. With a little bit of help from here we were able to implement the reversal without any bigger issues. Our main issue was that trying to implement a completely new `toString()` method was completely useless since the Node class already had a recursive method for doing just that implemented. So we just used this method.