INGI2132
Assignement 1 Report

Xavier Crochet, Julien De Coster

15 février 2013

1 Introduction

We implemented first the division, the unary plus and the modulo. We wrote likewise the program *primes.java* with the Eratosthenes algorithm. We used jUnit tests to test both the operators and the j– program.

2 Development

The main choice we did was on the data structure of the *primes.java* program. We analyzed two possibilies :

1. A structure with two lists. The first list contains integers from 1 to n. The second list contains the marks. Both lists are equals in size in order to match each element from the first list to a mark from the second one. An element is marked as 0 if it is potentially a prime number or as 1 if it is identified as a multiple of a previous number of the list. We use the term *potentially* because all the elements of the mark list are initialized at 0 except 0 and 1, initialized at 1.
2. A structure with an unique linked list. Here the strategy is to remove every element identified as a multiple of a previous number. The benefit of this method is clear : we don't have to visit elements wich are not relevant because already identied.

We choose the first strategy mainly for performance. Accessing elements and remove them from a linked list is costly. Our mesures showed that the first strategy took more or less ten times less time with wide values (e.g. 50 000).

3 Testing

3.1 Test strategy for the operators

3.1.1 Normal behaviour of the operators

The test here are straightforward. We have to check wether the operation behave normaly or not.
E.G.

– 4 / 3 = 1

- 4 % 3 = 1
- +1 = 1
- ...

3.1.2 Priority of the operators

Because we are dealing with arithmetic operatos, we have to check whether these operator respect the priority of the operation. Basicly, we implements rightPriority and leftPriority function in pass/*.java classes. Those one simply *mix* the concerned operation with the binary plus. For example

- 10 + 3 / 3 = 11 (and not 4!!!)
- 3 / 3 + 10 = 11 (and not 0!!!)
- ...%...

3.1.3 Exception rise

Concerning the modulo and the divisor operator, we have to test also that these operations rise the corrects exceptions whenever we try to divide by 0. Sadly, it's not possible to check wheter an exception is raised with the 3th version of junit. The workaround is to surround the concerned operation (int the concerned pass/*.java class) with a try catch instruction, call the fail() function just after the operation to force the programm to stop and to check in the catch block that the message of the raised exception is the good one.
E.G. : divide(3, 0)

```
try {
        division.divide(42, 0);
        fail();
}
catch (Exception e) {
        assertEquals("/ by zero", e.getMessage());
}
```

I know, it's nasty :-/

3.2 Test strategy for Primes

We try to think differently. We try to remember what the course LINGI1122(*Methode de conception de programme*). I.e. Be defensive, check arrays boundaries when dealing with and check extremes values. Concretly, we tested the programm with negatives value, *strange* value as 0, 1 and 2.

4 Conclusion

Trough implementing some simple feature into j–, it's clear that creating a programming langage from scratch is not the easiest thing to do.However, this first assignment show us an insight of *How to do it*. At last, it reminded us the good practice of programm developpement while testing our functions with Junit.