# Lab63: white box testing, unit

Goal of this lab is to practice white box testing of small software modules.

1. Define the test cases in high level format (ex tables), using WB techniques (statement coverage, decision coverage, …)
2. For Ex2 and Ex3: compare test cases written using BB techniques, and test cases written using BB techniques. Compare the WB coverage obtained by test cases written using BB techniques.
3. For Ex4, Ex5, Ex6. Write WB test cases in high level format,
4. For Ex4, Ex5, Ex6. Translate test cases written in point 3 in code format (JS + Jest). Run the test cases and verify the coverage computed by Jest.

**Ex 1 Max absolute**

**public** **int** max\_absolute(**int**[] numbers){

**if**(numbers.length > 5)

**return** -1;

**int** max\_value = 0;

**for**(**int** i = 0; i<numbers.length; i++){

**if** (numbers[i] < 0 )

max\_value = Math.*max*(max\_value,Math.*abs*(numbers[i]));

**else** max\_value = Math.*max*(max\_value, numbers[i]);

}

**return** max\_value;

}

This function should return the max absolute of an array of integers, containing at most 5 elements. Returns -1 in case of error.

Define test cases for statement coverage, decision coverage, loop coverage, path coverage (if possible).

**Ex 2 Convert Int**

A function converts a sequence of chars in an integer number. The sequence can start with a ‘-‘ (negative number). If the sequence is shorter than 6 chars, it is filled with blanks (to the left side). The integer number must be in the range minint = -32768 to maxint = 32767.The function signals an error if the sequence of chars is not allowed.

1. **public** **class** ConvertInt {
2. **public** **int** convert(**char**[] str) **throws** Exception{
3. **if** (str.length > 6)
4. **throw** **new** Exception();
5. **int** number=0;**int** digit; **int** i=0;
6. **if** (str[0]=='-')
7. i=1;
8. **for**(; i<str.length; i++){
9. digit = str[i] - '\0';
10. number = number \* 10 + digit;
11. }
12. **if** (str[0]=='-')
13. number = -number;
14. **if** (number > 32767 || number < -32768)
15. **throw** **new** Exception();
16. **return** number;
17. }
18. }

Define test cases for statement coverage, decision coverage, multiple condition coverage (line 14), loop coverage, path coverage (if possible).

Compare your test cases with the ones written with black box testing (Lab 5).

**Ex 3 Queue of events**

A queue of events in a simulation system receives events. Each event has a time tag. It is possible to extract events from the queue, the extraction must return the event with lower time tag. The queue discards events with negative or null time tag.The queue must accept at least 100.000 events.Events with the same time tag must be merged (i.e. the second received is discarded)

Define test cases for statement coverage, decision coverage, loop coverage, path coverage (if possible). Compare your test cases with the ones written with black box testing

1. **import** java.util.Iterator;
2. **import** java.util.LinkedList;
3. **public** **class** EventsQueue {
4. **private** LinkedList queue;
6. **public** EventsQueue(){
7. queue = **new** LinkedList();
8. }
9. **public** **void** insert(**int** event){
10. **int** index = 0;
11. **int** size = queue.size();
12. **while** (index < size &&
13. ((Integer)queue.get(index)).intValue() < event){
14. index++;
15. }
16. queue.add(index, **new** Integer(event));
17. }
18. **public** **int** pop(){
19. Object o = queue.getFirst();
20. **if** (o != **null**)
21. **return** ((Integer) o).intValue();
22. **else** **return** -1;
23. }
24. **public** **void** print(){
25. Iterator i = queue.iterator();
26. **int** event;
27. **while** (i.hasNext()){
28. event = ((Integer) i.next()).intValue();
29. System.out.println(event + " ");
30. }
31. }}

**Ex 4 Acceptable to eat**

The function *acceptableToEat* receives the weight in grams of, respectively, carbohydrates, proteins, fats in a serving of food. It returns true if

- the total amount of calories is < 1000

- (carb + protein) / fat > ½

class Example4{

static acceptableToEat( carb, protein, fat){

if(carb < 0 || protein < 0 || fat <0) return false;

if (4\*carb + 4\*protein + 9\*fat < 1000 && ((carb + protein) / fat > 0.5)) return true;

else return false;

}}

**Ex 5 BikeRentalFee**

This function computes (in euros) the fee for a bicycle rental, using these parameters

* duration: minutes the bicycle has been used
* minRate: cost per minute, in cents of euro
* minRate2: cost per minute, in cents of euro

The fee is computed as follows: free the first 30 minutes. minRate per min for the first hour exceeding the first 30 min (30 to 90 minutes), minRate2 after 90 minutes

Class Example5{

static computeFee(duration, minRate, minRate2) {

if (duration < 0 || minRate < 0 || minRate2 < 0) return -1.0;

if (duration < 30) return 0.0;

if (duration >= 30 && duration < 90) return minRate\*(duration - 30);

else return 60\*minRate + (duration-90)\*minRate2;

}}

**Ex 6 Railway ticket**

A railway company offers the possibility to people under 15 to travel free. The offer is dedicated to groups from 2 to 5 people traveling together.

For being eligible to the offer, at least a member of the group must be at least 18 years old. If this condition applies, all the under 15 members of the group travel free, and the others pay the Base Price.

The function computeFee receives as parameters basePrice (the price of the ticket), n\_passengers (the number of passengers of the group), n\_over18 (the number of passengers at least 18 old), n\_under15 (the number of passengers under 15 years old). It gives as output the amount that the whole group has to spend. It gives an error if groups are composed of more than 5 persons.

Class example6{

static computeFee(basePrice, n\_passengers, n\_over18, n\_under15) {

if (basePrice < 0 || n\_passengers < 0 || n\_over18 < 0 || n\_under15 < 0) return -1.0;

if (n\_over18 + n\_under15 > n\_passengers) return -1.0;

if (n\_passengers >=2 && n\_passengers <=5 && n\_over18>0) {

return basePrice\*(n\_passengers -n\_under15);

}

else {

return basePrice\*n\_passengers;

}

}

}}

**Ex 7 Tax Calculator**

**public** **double** elonianTaxCalculator(**double** income, **int** nDependents) {

**double** TaxSubTotal,Exemption,TaxTotal;

// first if - check income

**if** (income < 0) {

System.*out*.println("You cannot have a negative income.\n");

**return** -1;

}

// second if - check dependents

**if** (nDependents <= 0) {

System.*out*.println("You must have at least one dependent.\n");

**return** -2;

}

// third if (else-if) - compute tax subtotal

**if** (income < 10000)

TaxSubTotal = .02 \* income;

**else** **if** (income < 50000)

TaxSubTotal = 200 + .03 \* (income - 10000);

**else**

TaxSubTotal = 1400 + .04 \* (income - 50000);

Exemption= nDependents \* 50;

TaxTotal=TaxSubTotal - Exemption;

// last if - check negative tax

**if** (TaxTotal<0) //In case of negative tax

TaxTotal=0;

System.*out*.println( "$S$S$S$S$S$S$S$S$S$S$S$S$S$S$S$S$S$ \n");

System.*out*.println( "Elbonian Tax Collection Agency \n");

System.*out*.println( "Tax Bill \n");

System.*out*.println( "Citizen's Income: " + income +'\n');

System.*out*.println( "Tax Subtotal: " + TaxSubTotal +'\n');

System.*out*.println( "Number of Dependents: "+ nDependents + '\n');

System.*out*.println( "Tax Exepmtion: " + Exemption +'\n');

System.*out*.println( "Final Tax Bill: " + TaxTotal + '\n');

System.*out*.println( "$S$S$S$S$S$S$S$S$S$S$S$S$S$S$S$S$S$ \n");

**return** TaxTotal;

}

Define test cases for statement coverage, decision coverage, path coverage (if possible).

**Ex 8**

Consider the following code:

1. **float** foo (**int** a, **int** b, **int** c, **int** d, **float** e) {
2. **float** e;
3. **if** (a == 0) {
4. **return** 0;
5. }
6. **int** x = 0;
7. **if** ((a==b) || ((c == d) && bug(a) )) {
8. x=1;
9. }
10. e = 1/x;
11. **return** e;
12. }

Function bug(a) should return a value of true when passed a value of a=1.

Define test cases for statement coverage, decision coverage, multiple condition coverage, path coverage (if possible).