Åbo Akademi University

Software Testing

Assignment 1

Luis Araújo(2004624)

April 14, 2021

Contents

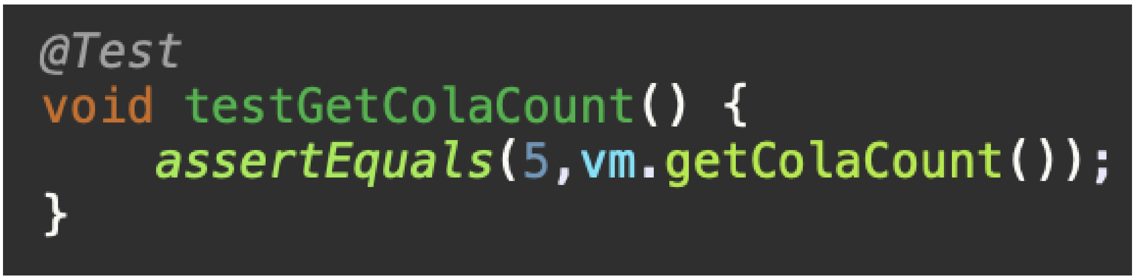
Chapter 1

Task 1

1.1 Write one jUnit test for each public method (except main()) and report code coverage of the tests per method

To complete this task, i started by creating test for the get methods :

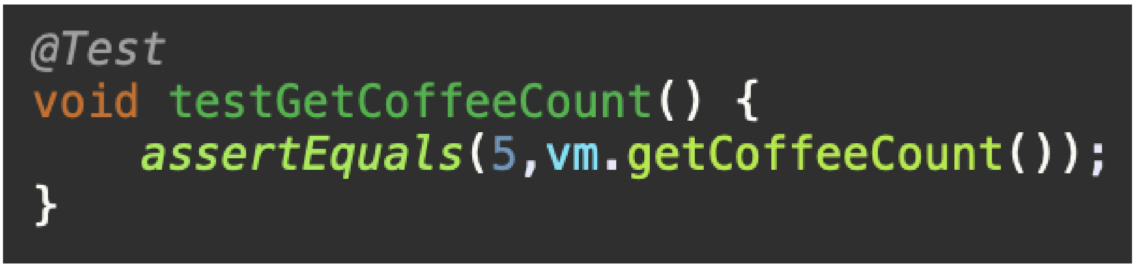
- **getColaCount**



```
@Test
void testGetColaCount() {
    assertEquals(5,vm.getColaCount());
}
```

Figure 1.1: getColaCount Test

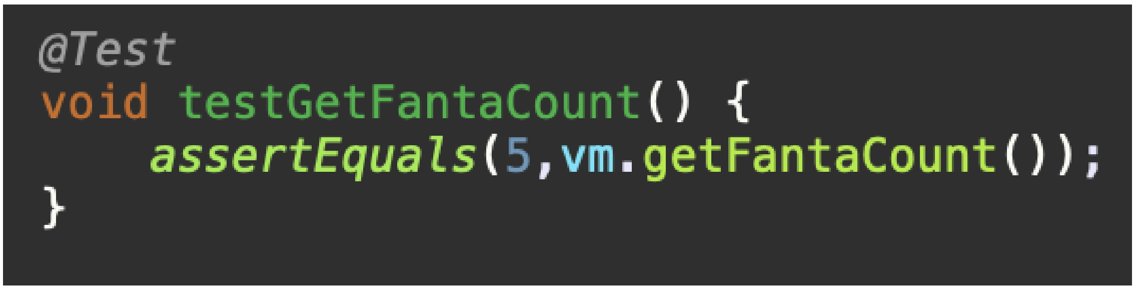
- **getCoffeeCount**



```
@Test
void testGetCoffeeCount() {
    assertEquals(5,vm.getCoffeeCount());
}
```

Figure 1.2: getCoffeeCount Test

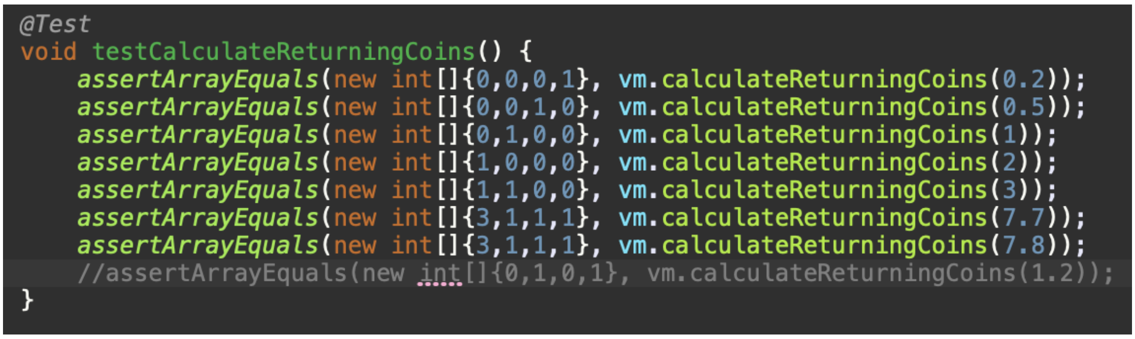
- **getFantaCount**



```
@Test
void testGetFantaCount() {
    assertEquals(5,vm.getFantaCount());
}
```

Figure 1.3: getFantaCount Test

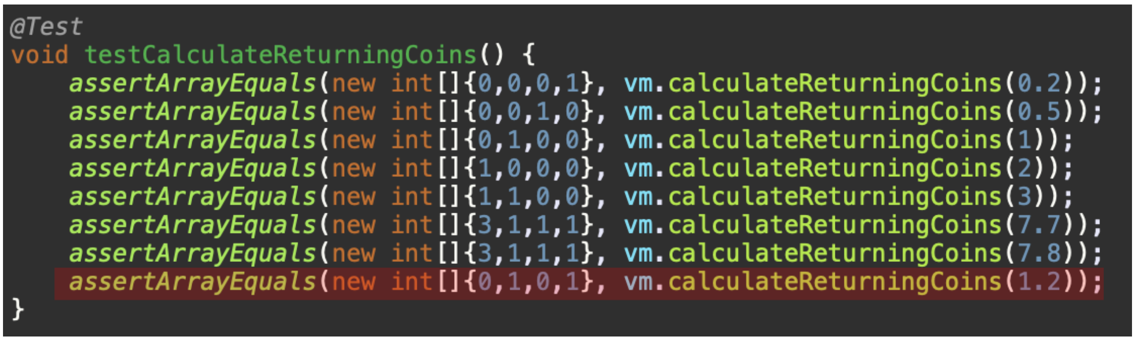
Continuing with the test of the methods, the next one to be tested was the **calculateReturningCoins** method. Which one, two different errors were identified, by examining the figure 1.4 we can see that two different inputs (the 7.7 and the 7.8 input) of the function calculateReturningCoins not only gives the same output, but the 7.8 gives the wrong one, this output should be [3,1,0,4] and not [3,1,1,1].



```
@Test
void testCalculateReturningCoins() {
    assertArrayEquals(new int[]{0,0,0,1}, vm.calculateReturningCoins(0.2));
    assertArrayEquals(new int[]{0,0,1,0}, vm.calculateReturningCoins(0.5));
    assertArrayEquals(new int[]{0,1,0,0}, vm.calculateReturningCoins(1));
    assertArrayEquals(new int[]{1,0,0,0}, vm.calculateReturningCoins(2));
    assertArrayEquals(new int[]{1,1,0,0}, vm.calculateReturningCoins(3));
    assertArrayEquals(new int[]{3,1,1,1}, vm.calculateReturningCoins(7.7));
    assertArrayEquals(new int[]{3,1,1,1}, vm.calculateReturningCoins(7.8));
    //assertArrayEquals(new int[]{0,1,0,1}, vm.calculateReturningCoins(1.2));
}
```

Figure 1.4: calculateReturnCoins Test

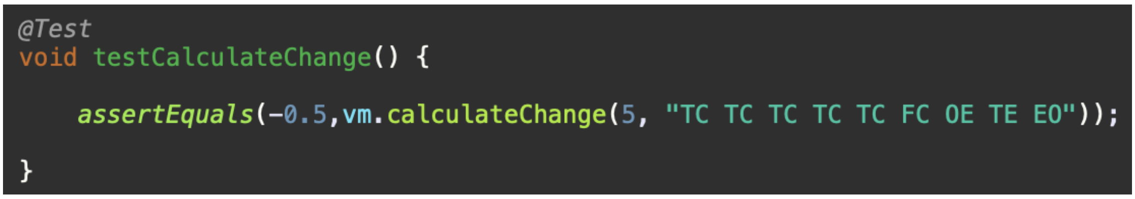
The other error identified, is the output of the 1.2 input, where the correct combination of coins should be [0,1,0,1] and the result of the test is a fail.



```
@Test
void testCalculateReturningCoins() {
    assertArrayEquals(new int[]{0,0,0,1}, vm.calculateReturningCoins(0.2));
    assertArrayEquals(new int[]{0,0,1,0}, vm.calculateReturningCoins(0.5));
    assertArrayEquals(new int[]{0,1,0,0}, vm.calculateReturningCoins(1));
    assertArrayEquals(new int[]{1,0,0,0}, vm.calculateReturningCoins(2));
    assertArrayEquals(new int[]{1,1,0,0}, vm.calculateReturningCoins(3));
    assertArrayEquals(new int[]{3,1,1,1}, vm.calculateReturningCoins(7.7));
    assertArrayEquals(new int[]{3,1,1,1}, vm.calculateReturningCoins(7.8));
    assertArrayEquals(new int[]{0,1,0,1}, vm.calculateReturningCoins(1.2));
}
```

Figure 1.5: calculateReturnCoins Error

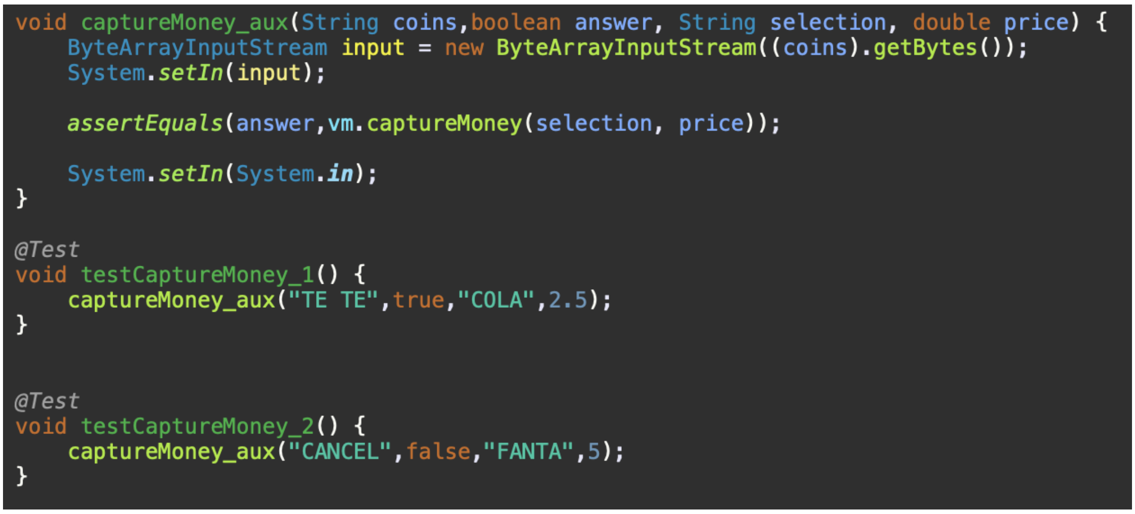
After, it was tested the **calculateChange** method, where, as the input of the function, it was given a negative balance of the purchase, all the different coins, and even a invalid coin.



```
@Test
void testCalculateChange() {

    assertEquals(-0.5,vm.calculateChange(5, "TC TC TC TC TC FC OE TE EO"));

}
```

Figure 1.6: calculateChange Test

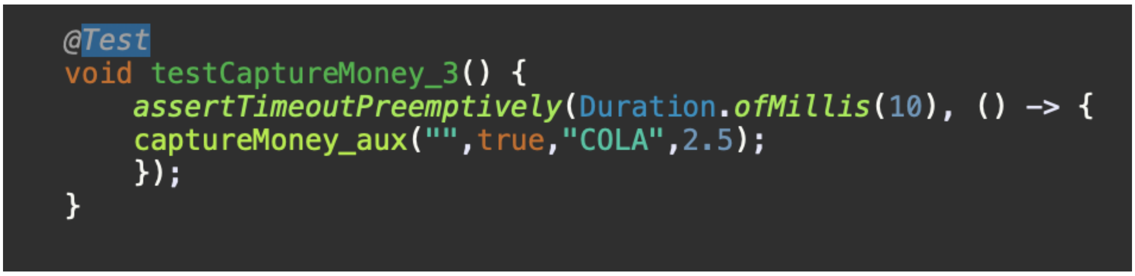
Following the testing of the method **captureMoney**, the approach had to be more thoughtful. In this one, since the method had a *while*(*true*) loop and also, and since it was necessary the input of the user, the process was a little more different.



```
void captureMoney_aux(String coins,boolean answer, String selection, double price) {
    ByteArrayInputStream input = new ByteArrayInputStream((coins).getBytes());
    System.setIn(input);

    assertEquals(answer,vm.captureMoney(selection, price));

    System.setIn(System.in);
}

@Test
void testCaptureMoney_1() {
    captureMoney_aux("TE TE",true,"COLA",2.5);
}


@Test
void testCaptureMoney_2() {
    captureMoney_aux("CANCEL",false,"FANTA",5);
}
```

Figure 1.7: captureMoney Test

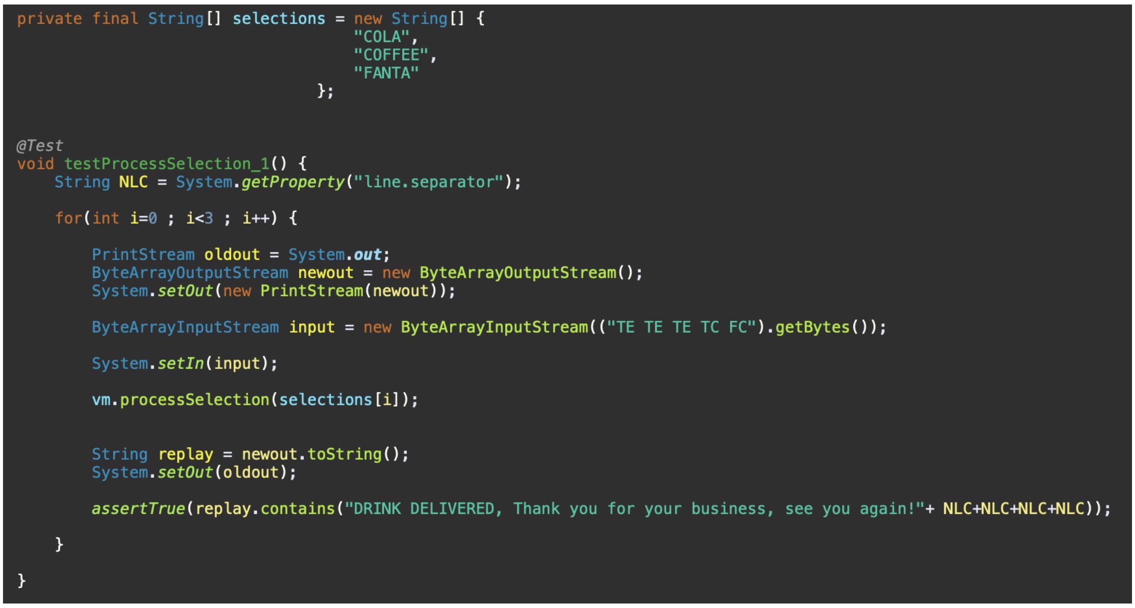
To have a better coverage of the program it was also needed to add this piece of code, even though this is a failed test.



```
@Test
void testCaptureMoney_3() {
    assertTimeoutPreemptively(Duration.ofMillis(10), () -> {
    captureMoney_aux("",true,"COLA",2.5);
    });
}
```

Figure 1.8: captureMoney better coverage Test

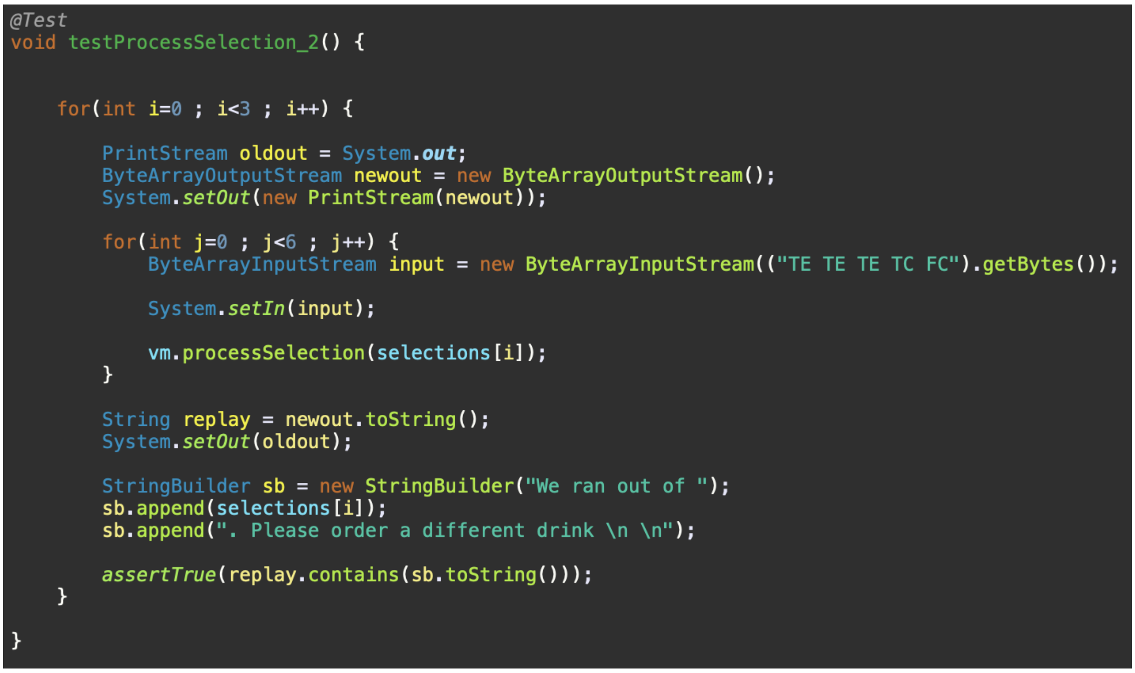
To test this next method, **processSelection**, I decided to slipt it in two different parts. The first part is represented by the figure 1.9, it enhances all the correct cases for each selection.



```
private final String[] selections = new String[] {
                                        "COLA",
                                        "COFFEE",
                                        "FANTA"
                                   };


@Test
void testProcessSelection_1() {
    String NLC = System.getProperty("line.separator");

    for(int i=0 ; i<3 ; i++) {

        PrintStream oldout = System.out;
        ByteArrayOutputStream newout = new ByteArrayOutputStream();
        System.setOut(new PrintStream(newout));

        ByteArrayInputStream input = new ByteArrayInputStream(("TE TE TE TC FC").getBytes());

        System.setIn(input);

        vm.processSelection(selections[i]);


        String replay = newout.toString();
        System.setOut(oldout);

        assertTrue(replay.contains("DRINK DELIVERED, Thank you for your business, see you again!"+ NLC+NLC+NLC+NLC));

    }

}
```

Figure 1.9: processSelection Test 1

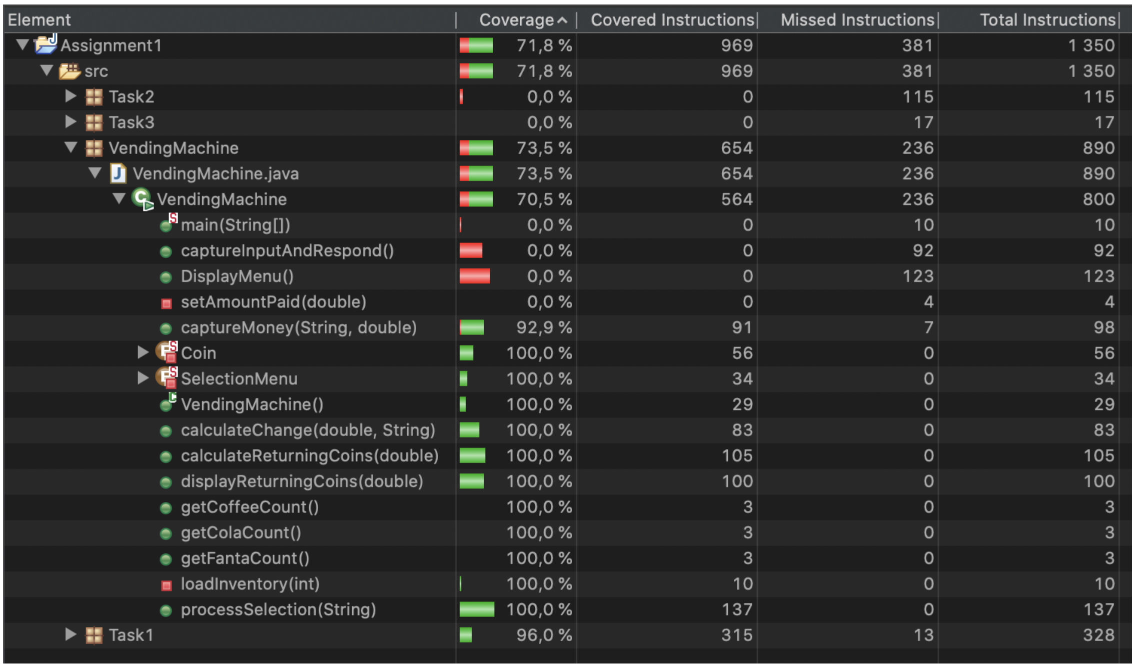
And, the second part, highlights all of the wrong cases of the method, which is when there is no more selections of a product in the vending machine.



```
@Test
void testProcessSelection_2() {


    for(int i=0 ; i<3 ; i++) {

        PrintStream oldout = System.out;
        ByteArrayOutputStream newout = new ByteArrayOutputStream();
        System.setOut(new PrintStream(newout));

        for(int j=0 ; j<6 ; j++) {
            ByteArrayInputStream input = new ByteArrayInputStream(("TE TE TE TC FC").getBytes());

            System.setIn(input);

            vm.processSelection(selections[i]);
        }

        String replay = newout.toString();
        System.setOut(oldout);

        StringBuilder sb = new StringBuilder("We ran out of ");
        sb.append(selections[i]);
        sb.append(". Please order a different drink \n \n");

        assertTrue(replay.contains(sb.toString()));
    }

}
```

Figure 1.10: processSelection Test 2

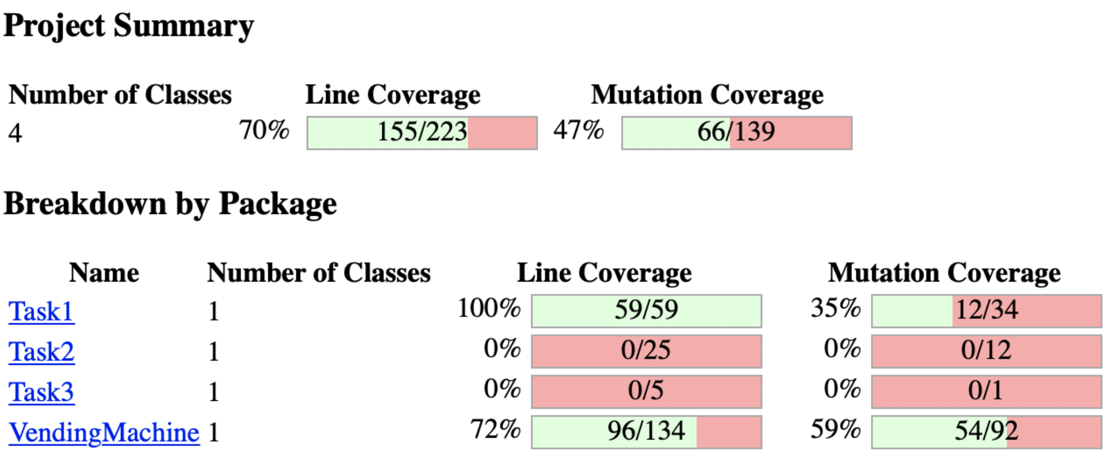
1.2 Line coverage



Element	Coverage	Covered Instructions	Missed Instructions	Total Instructions
Assignment1	71,8 %	969	381	1 350
src	71,8 %	969	381	1 350
Task2	0,0 %	0	115	115
Task3	0,0 %	0	17	17
VendingMachine	73,5 %	654	236	890
VendingMachine.java	73,5 %	654	236	890
VendingMachine	70,5 %	564	236	800
main(String[])	0,0 %	0	10	10
captureInputAndRespond()	0,0 %	0	92	92
DisplayMenu()	0,0 %	0	123	123
setAmountPaid(double)	0,0 %	0	4	4
captureMoney(String, double)	92,9 %	91	7	98
Coin	100,0 %	56	0	56
SelectionMenu	100,0 %	34	0	34
VendingMachine()	100,0 %	29	0	29
calculateChange(double, String)	100,0 %	83	0	83
calculateReturningCoins(double)	100,0 %	105	0	105
displayReturningCoins(double)	100,0 %	100	0	100
getCoffeeCount()	100,0 %	3	0	3
getColaCount()	100,0 %	3	0	3
getFantaCount()	100,0 %	3	0	3
loadInventory(int)	100,0 %	10	0	10
processSelection(String)	100,0 %	137	0	137
Task1	96,0 %	315	13	328

Figure 1.11: Line coverage

1.3 Mutation



Project Summary

Number of Classes	Line Coverage		Mutation Coverage	
4	70%	155/223	47%	66/139

Breakdown by Package

Name	Number of Classes	Line Coverage		Mutation Coverage	
Task1	1	100%	59/59	35%	12/34
Task2	1	0%	0/25	0%	0/12
Task3	1	0%	0/5	0%	0/1
VendingMachine	1	72%	96/134	59%	54/92

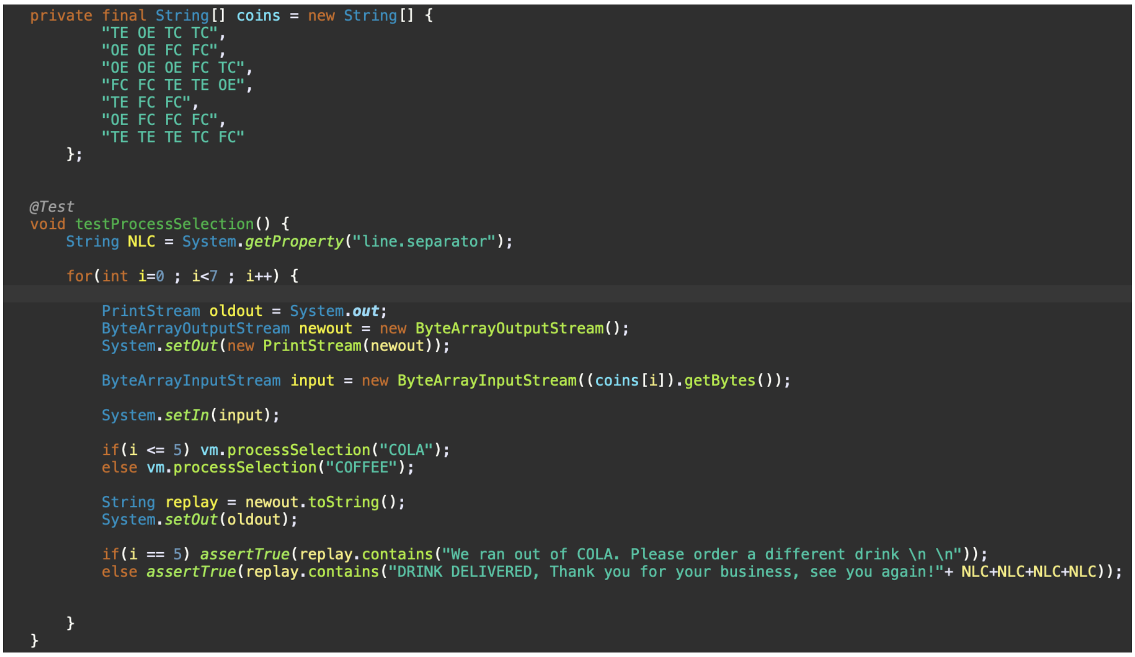
Figure 1.12: Mutation coverage

Chapter 2

Task 2

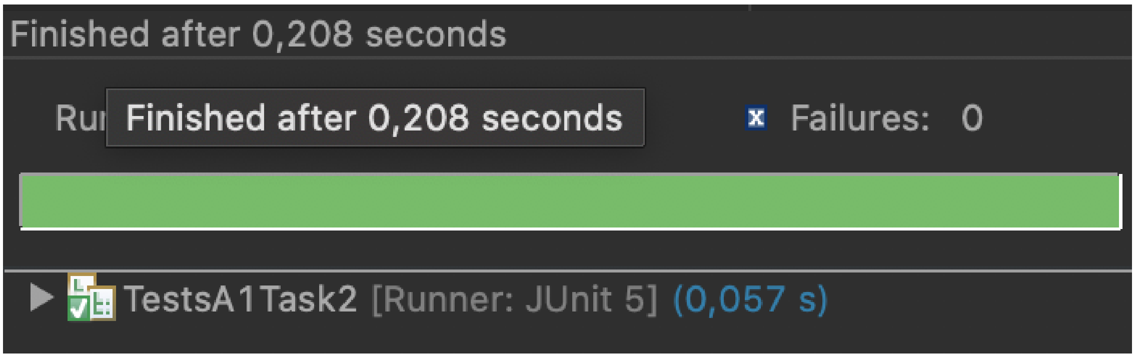
2.1 Write a jUnit test case that tests the following scenario: “the user tries to buy 6 COLA (only 5 available) and a COFFEE. For every drink a different combination of coins is used to pay”.

For this test, as we can see in the figure 2.1, all parameters of the description are followed, from the purchase a sixth COLA to the use of a different combination of coins.



```java
private final String[] coins = new String[] {
        "TE OE TC TC",
        "OE OE FC FC",
        "OE OE OE FC TC",
        "FC FC TE TE OE",
        "TE FC FC",
        "OE FC FC FC",
        "TE TE TE TC FC"
    };


@Test
void testProcessSelection() {
    String NLC = System.getProperty("line.separator");

    for(int i=0 ; i<7 ; i++) {

        PrintStream oldout = System.out;
        ByteArrayOutputStream newout = new ByteArrayOutputStream();
        System.setOut(new PrintStream(newout));

        ByteArrayInputStream input = new ByteArrayInputStream((coins[i]).getBytes());

        System.setIn(input);

        if(i <= 5) vm.processSelection("COLA");
        else vm.processSelection("COFFEE");

        String replay = newout.toString();
        System.setOut(oldout);

        if(i == 5) assertTrue(replay.contains("We ran out of COLA. Please order a different drink \n \n"));
        else assertTrue(replay.contains("DRINK DELIVERED, Thank you for your business, see you again!"+ NLC+NLC+NLC+NLC));


    }
}
```

Figure 2.1: Scenario test



Finished after 0,208 seconds

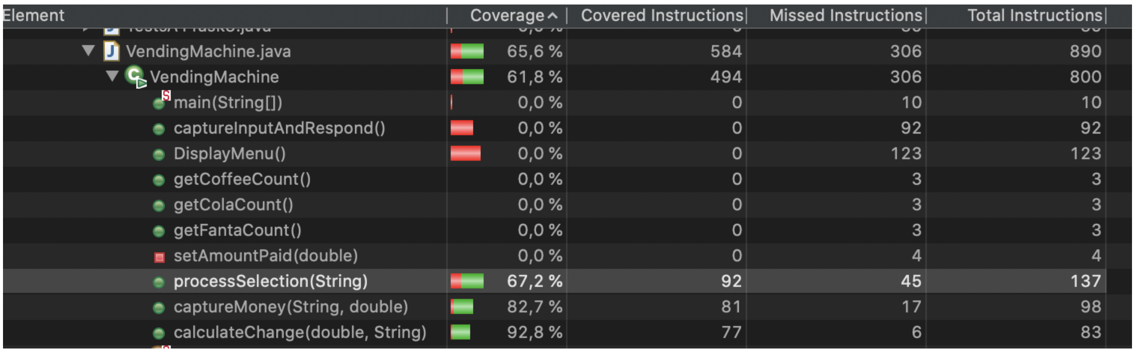
Ru Finished after 0,208 seconds ☒ Failures: 0

▶ TestsA1Task2 [Runner: JUnit 5] (0,057 s)

Figure 2.2: All tests passed

2.2 Line coverage

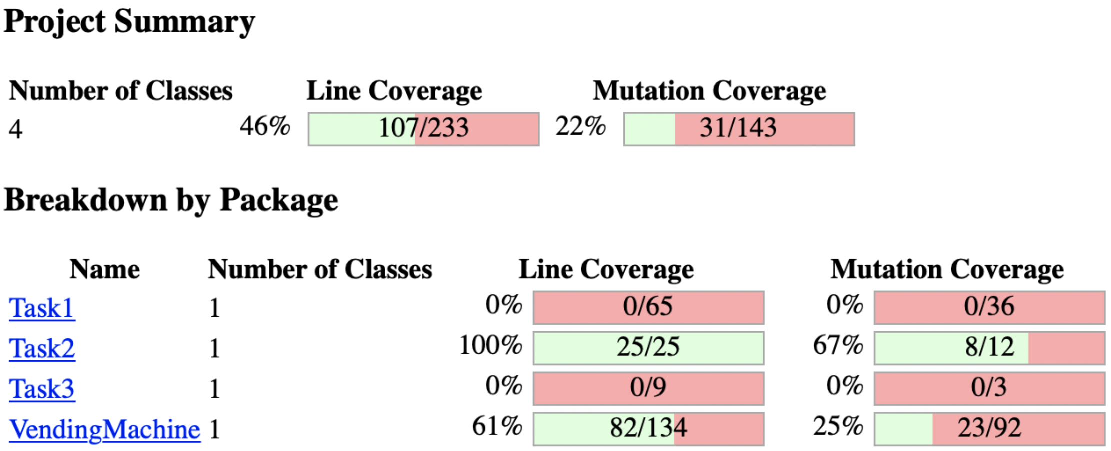
From this scenario, 67.2% of the method **processSelection** is covered.



Element	Coverage	Covered Instructions	Missed Instructions	Total Instructions
▼ VendingMachine.java	65,6 %	584	306	890
▼ VendingMachine	61,8 %	494	306	800
main(String[])	0,0 %	0	10	10
captureInputAndRespond()	0,0 %	0	92	92
DisplayMenu()	0,0 %	0	123	123
getCoffeeCount()	0,0 %	0	3	3
getColaCount()	0,0 %	0	3	3
getFantaCount()	0,0 %	0	3	3
setAmountPaid(double)	0,0 %	0	4	4
processSelection(String)	67,2 %	92	45	137
captureMoney(String, double)	82,7 %	81	17	98
calculateChange(double, String)	92,8 %	77	6	83

Figure 2.3: Coverage of the test

And 67% of the mutation is covered.



Project Summary

Number of Classes	Line Coverage		Mutation Coverage	
4	46%	107/233	22%	31/143

Breakdown by Package

Name	Number of Classes	Line Coverage		Mutation Coverage	
Task1	1	0%	0/65	0%	0/36
Task2	1	100%	25/25	67%	8/12
Task3	1	0%	0/9	0%	0/3
VendingMachine	1	61%	82/134	25%	23/92

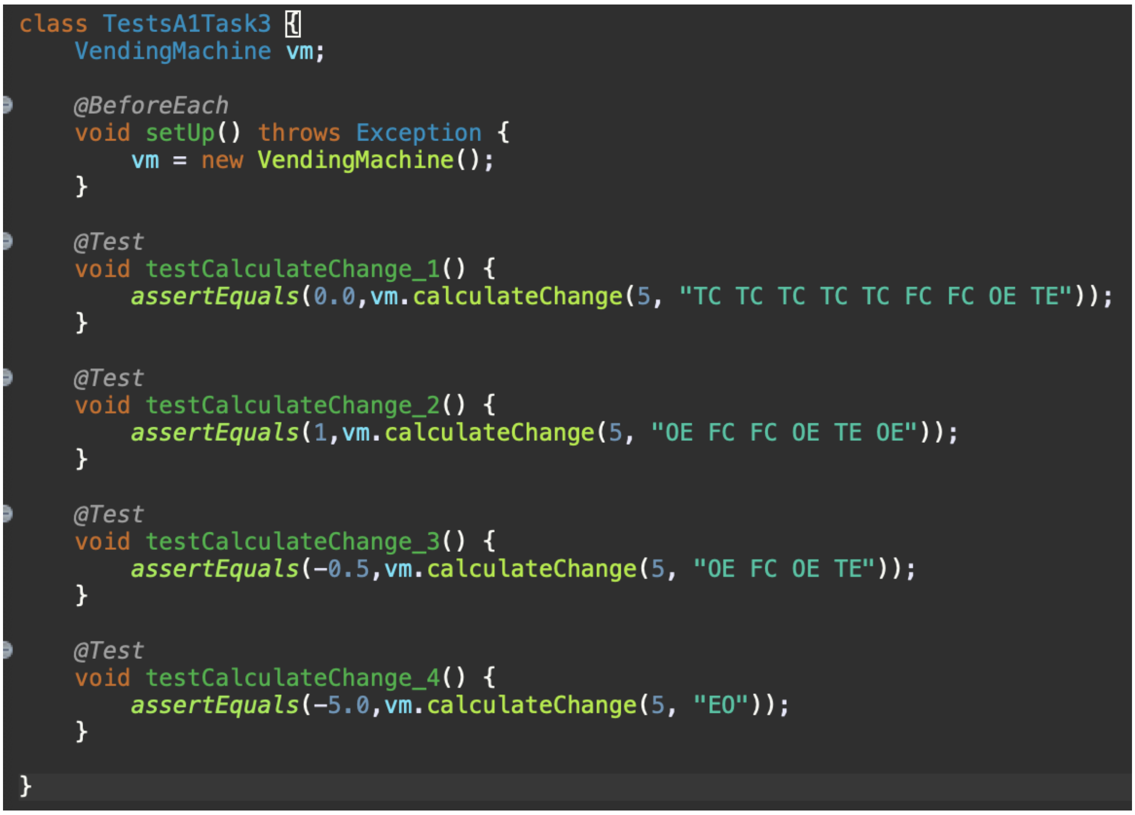
Figure 2.4: Mutation coverage of the test

Chapter 3

Task 3

3.1 Write jUnit tests for public double calculateChange(double price, String insertedCoins)

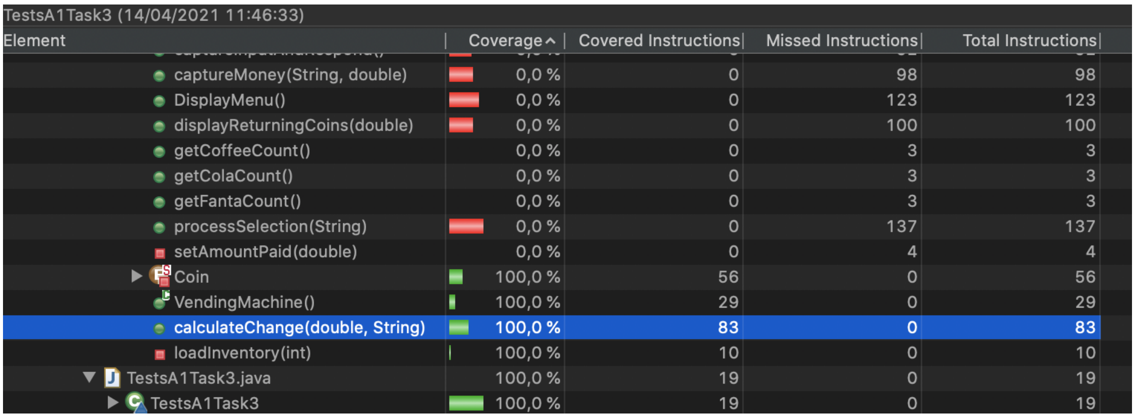
Finally, for this last task it was tested the **calculateChange** method to try to obtain 100% of line coverage and 100% of mutation coverage. So, trying to get the maximum live coverage, it was tested all the possibilities of outcome from the method, when the output is positive, negative, zero, and when its provided a wrong coin.



```
class TestsA1Task3 {
    VendingMachine vm;

    @BeforeEach
    void setUp() throws Exception {
        vm = new VendingMachine();
    }

    @Test
    void testCalculateChange_1() {
        assertEquals(0.0,vm.calculateChange(5, "TC TC TC TC TC FC FC OE TE"));
    }

    @Test
    void testCalculateChange_2() {
        assertEquals(1,vm.calculateChange(5, "OE FC FC OE TE OE"));
    }

    @Test
    void testCalculateChange_3() {
        assertEquals(-0.5,vm.calculateChange(5, "OE FC OE TE"));
    }

    @Test
    void testCalculateChange_4() {
        assertEquals(-5.0,vm.calculateChange(5, "EO"));
    }

}
```

Figure 3.1: calculateChange Test

Just as requested the line coverage is 100%.

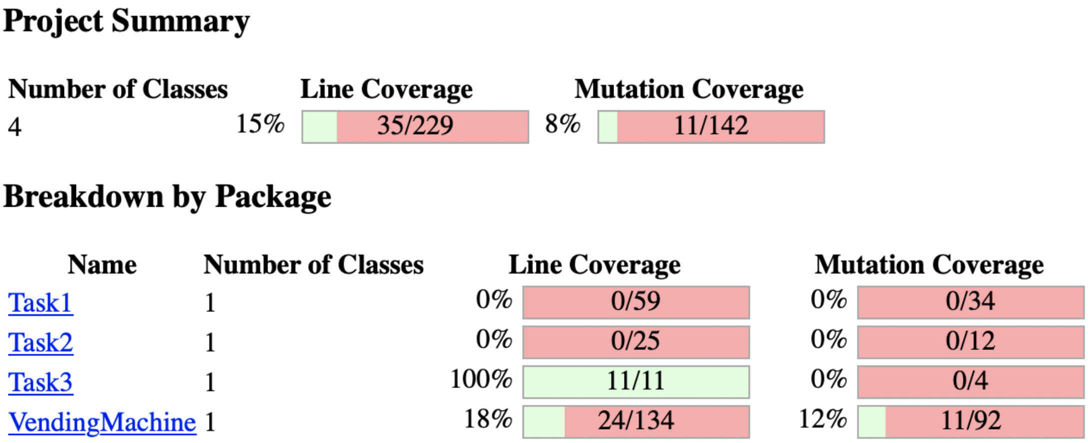


TestsA1Task3 (14/04/2021 11:46:33)

Element	Coverage	Covered Instructions	Missed Instructions	Total Instructions
captureMoney(String, double)	0,0 %	0	98	98
DisplayMenu()	0,0 %	0	123	123
displayReturningCoins(double)	0,0 %	0	100	100
getCoffeeCount()	0,0 %	0	3	3
getColaCount()	0,0 %	0	3	3
getFantaCount()	0,0 %	0	3	3
processSelection(String)	0,0 %	0	137	137
setAmountPaid(double)	0,0 %	0	4	4
Coin	100,0 %	56	0	56
VendingMachine()	100,0 %	29	0	29
calculateChange(double, String)	100,0 %	83	0	83
loadInventory(int)	100,0 %	10	0	10
TestsA1Task3.java	100,0 %	19	0	19
TestsA1Task3	100,0 %	19	0	19

Figure 3.2: Line coverage

But unfortunately I wasn't able to obtain any mutation coverage.



Project Summary

Number of Classes	Line Coverage		Mutation Coverage	
4	15%	35/229	8%	11/142

Breakdown by Package

Name	Number of Classes	Line Coverage		Mutation Coverage	
Task1	1	0%	0/59	0%	0/34
Task2	1	0%	0/25	0%	0/12
Task3	1	100%	11/11	0%	0/4
VendingMachine	1	18%	24/134	12%	11/92

Figure 3.3: Line coverage