Software Testing Lab 1 Unit Tests

1. Name Vitalij Hein
2. MatrNr 11932447
3. LVA-Titel Software Testing
4. LVA-Nr 183.290
5. Datum 29.10.19

Which bugs did I encounter during the test implementation? What was their behavior and how did I fix them?

1. The very first bug I found in the code was the following one:

```
if(requiredStorage > inbox.length || (inbox.length - occupied) <= requiredStorage) {
    throw new NotEnoughSpaceException("Storage Overflow");
}
```

What was its behavior?

The code was not able to write to the MobileStorage even if there was enough storage size available.

The problem was the following:

Preset:

1. We have three messages.
2. The storageSize is set to three.
3. Trying to save all three messages to the storage, which is set to three.
4. Saving the first works because:
    a. requiredStorage (1) is smaller than the inbox length (3). [TRUE]
    b. Inbox length (3) – occupied (0) is not smaller then or equal to requiredStorage(1) [TRUE]
5. Saving the second message works fails:
    a. requiredStorage (1) is smaller than the inbox length (3). [TRUE]
    b. Inbox length (3) – occupied (1) is not smaller or equal to requiredStorage(1) [TRUE]
6. Saving the second message works fails:
    a. requiredStorage (1) is smaller than the inbox length (3). [TRUE]
    b. Inbox length (3) – occupied (2) is not smaller or equal to requiredStorage(1) [FALSE]

```
-> throw new NotEnoughSpaceException("Storage Overflow");
```

So even if there is still free storage available, the program doesn‘t recognize it. To change this behavior, we can simple swap the „<=“ to „<“ like in this:

```
if (requiredStorage > inbox.length || (inbox.length - occupied) < requiredStorage) {
    throw new NotEnoughSpaceException("Storage Overflow");
}
```

What it does?

If we look back to 6. In the description above, we will see that:

6. Saving the second message works fails:
    a. requiredStorage (1) is smaller than the inbox length (3). [TRUE]
    b. Inbox length (3) – occupied (2) is not smaller than requiredStorage(1) [TRUE]

➔ NO NotEnoughSpaceException!

2. The second bug I found was an ArrayOutOfBoundsException in the second line below.

```
IntStream.range(1, occupied).forEach(index -> inbox[index-1] = inbox[index]);
inbox[occupied] = null;
```

What was its behavior?

The test „deleteMessage_CheckIfMessageCanBeDeletedIfOnly1Message()“ was not able to save a message and afterwards to delete the same messages, if the MobileStorage was set to one.

```
@Test
public void deleteMessage_CheckIfMessageCanBeDeletedIfOnly1Message() {
    MobileStorage mobileStorage = new MobileStorage( storageSize: 1);
    mobileStorage.saveMessage("message");
    mobileStorage.deleteMessage();
}
```

The problem was the following: The size of the inbox was set to 1 by defining MobileStorage(1). Because inbox[] is an array of the size of 1, we can only access the first index with inbox[0]. So, when trying to access inbox[occupied] the program reads inbox[1], which would be an array with the length of 2.

That’s why the program would terminate with the ArrayOutOfBoundsException.

How did I change it?

I solved it by looking at the only array in the deleteMessage-Method. After spotting inbox[occupied] I decided to subtract a single digit from occupied so that it would match the array length.

```
IntStream.range(1, occupied).forEach(index -> inbox[index - 1] = inbox[index]);
inbox[occupied-1] = null;
```

3. The third error encountered was a NullPointerException.

I found the error again in the method „deleteMessage“.

```
inbox[0].setPredecessor(null);
occupied--;
```

The behavior:

Again, deleting a message straight after creating it, while the MobileStorage is set to 1 leads to an exception. This time the following happens:

While deleting the message we set the inbox[occupied-1], which means inbox[0] to null in the example above. After setting it to null, we clearly have no object in the inbox stored.

But still we try to access the MobileMessage object in storage, because we try to set the predecessor to null. Not only it's redundant in this case, it's even not working.

How did I change it?

```
if(inbox[0] != null){
inbox[0].setPredecessor(null);}
```

I just added a simple check if inbox[0] is not null, if so, the „setPredecessor" method is never called.

4. The last bug was found in the method „getLastMessage".

The implementation always leads to a NullPointerException.

```
List<MobileMessage> successors = Arrays.stream(inbox)
        .filter(msg -> msg != null && msg.getPredecessor().equals(message))
        .collect(Collectors.toList());
```

The behavior:

The implementation always leads to a NullPointerException. That's because we pass a message when calling the „getLastMessage" method. And if the MobileStorage is set to 100, but we have for example only three messages saved, the we have 3 „real" messages and 97 „null"-objects saved. So every time we trying to pass a null the code fails.

How did I change it?

```
List<MobileMessage> successors = Arrays.stream(inbox)
        .filter(msg -> msg != null && msg.getPredecessor() == message)
        .collect(Collectors.toList());
```

I changed „equals", which checks if the objects are equals to „==" which checks if the object reference is the same. And because "equals" is a method, it will get you a NullPointerException, if you try to invoke it on a null-reference.

Which of my test failed initially due to bugs in the implementation?

1. saveMessage_CheckIfStorageSizeIsActuallyMessageSize()
2. deleteMessage_CheckIfMessageCanBeDeletedIfOnly1Message()
3. deleteMessage_CheckIfMessageCanBeDeletedIfOnly1Message()
4. searchOccurrences_CheckIfSearchCriteriaIsFound()

Discuss your coverage report (Is the coverage sufficient? Why (not)? Which coverage value should be reached and why?)

The overall coverage summary (please the in the zip file) tells us, that the written tests are covering 100% of the classes , 66.7% of the methods of the classes and 82,4% of the lines of all classes in all packages.

Digging deeper we can see that in the package „at.ac.tuwien.inso.peso“ we have a code coverage of :

1. 100% in classes
2. 100% in methods
3. 100% in lines

But the packages „at.ac.tuwien.inso.peso.exception“ tells a that:

1. 100% in classes
2. 20% in methods
3. 20% in lines

So what's happening here?

First of all I think it's a good sign that the main package „at.ac.tuwien.inso.peso“ returns a full value of 100%. That indicates, at least the test written are able to access every part of the code in this package. This means we have no dead code.

But also I don't think that the package „at.ac.tuwien.inso.peso.exception“ should have a coverage of 100%. First, we don't even call all of the exceptions created there in the implementation. That means we have dead code here. Dead code should be removed. Even though in a future theoretical development they could be used, I will delete them. Most of the time we are working with a version control software, which can always retrieve a method we deleted in the past.

I won't delete them, only to show that we don't need 100% code coverage to have a set of some good stable tests. (Hope this line was not to arrogant 😊)

Should we always strive for 100% code coverage? Yes and no. Yes, because it's a good indicator if we have dead code and No, because 100% code coverage doesn't mean, we have written every possible test. It only tells us: We reached every possible line/method/class at least once!

Mutation Test and Discussion All in One.

After assuring that every test case is green, I started the first mutation test.

On the first run I had the following stats:

Pit Test Coverage Report

Package Summary

at.ac.tuwien.inso.peso

Number of Classes	Line Coverage		Mutation Coverage	
2	99%	69/70	82%	37/45

Breakdown by Class

Name	Line Coverage		Mutation Coverage	
MobileMessage.java	100%	8/8	100%	2/2
MobileStorage.java	98%	61/62	81%	35/43

So, Pit Test generated 45 mutations of which I was able to cover 37. Also, we can see the main mutations were generated in MobileStorage.java. That's because in the MobileMessage.java are only getter and setter method.

After digging deeper, I was able to see the following:

```
29      public MobileStorage(int storageSize) {
30  2       if (storageSize < 1) {
31              throw new IllegalArgumentException("Storage size must be greater than 0");
32          }
```

```
1. changed conditional boundary → SURVIVED
```

What is happening here?

The mutation test simply creates this case: if (storageSize <=1)

To kill this mutation I created the following test:

```
@Test(expectedExceptions = StorageEmptyException.class)
public void deleteMessage_CheckIfErrorIsThrownWhileOccupiedZero() {
    MobileStorage mobileStorage = new MobileStorage( storageSize: 1);
    mobileStorage.deleteMessage();
}
```

With this test we assure that we will notice if the conditional boundary will be changed by other programmers or by evil attackers.

The next survived mutation were:

```
    1. changed conditional boundary → SURVIVED
    2. changed conditional boundary → SURVIVED
52  3. Replaced integer subtraction with addition → SURVIVED
```

The first two were also killed afterwards with the above test.

The third was killed afterwards by this killer:

```
@Test(expectedExceptions = NotEnoughSpaceException.class)
public void saveMessage_CheckRequiredStorageCalculationWithMultipleMessages() {
    String message1 = "This";
    String message2 = "Booooom.";
    MobileStorage mobileStorage = new MobileStorage( storageSize: 1);
    mobileStorage.saveMessage(message1);
    mobileStorage.saveMessage(message2);

}
```

Then we had the following survivors:

```
89  1. removed call to java/util/stream/IntStream::forEach → SURVIVED
    2. Replaced integer subtraction with addition → SURVIVED
```

Both were killed afterwards by creating this test:

```java
//(bug deleter)
@Test
public void deleteMessage_CheckInboxAfterDeletion() {
    MobileStorage mobileStorage = new MobileStorage( storageSize: 10);
    mobileStorage.saveMessage("Test-String");
    mobileStorage.saveMessage("Test-String 2");
    mobileStorage.saveMessage("Test-String 3");
    mobileStorage.deleteMessage();
    String actualDeletedMessage = mobileStorage.listMessages();
    String expectedDeletedMessage = "Test-String 2\nTest-String 3";
    Assert.assertEquals(actualDeletedMessage, expectedDeletedMessage);


}
```

And the last two survivors.

```
91   1. removed call to at/ac/tuwien/inso/peso/MobileMessage::setPredecessor → SURVIVED
92   1. Replaced integer subtraction with addition → SURVIVED
```

Now it's getting interesting. While I managed to kill the survivor in line 92 with this:

```java
@Test
public void deleteMessage_CheckIfOccupiedIsCorrect() {
    MobileStorage mobileStorage = new MobileStorage( storageSize: 10);
    mobileStorage.saveMessage("message");
    mobileStorage.deleteMessage();
    int actualOccupied = mobileStorage.getOccupied();
    int expectedOccupied = 0;
    Assert.assertEquals(actualOccupied, expectedOccupied);
}
```

I found no way to kill the survivor in line 91.

My first attempt was by using reflections to access the private MobileMessage inbox[] variable.

But first it didn't work and second it's bad practice. Unit Test should be easy written test codes and never more complicated than the to be tested code.

My second idea was: Why not setting the variable public. But honestly setting my inbox on my smartphone device on public doesn't seem to be a good problem. It's private for a good reason. And also, unit testing should focus on behavior rather than implementation.

The third option would be to create a getter. But here would be the problem that getters are always returning a new instance of the variable, instead of getting the same one used in the MobileStorage class.

So that's why I decided not to touch that piece of code and let the mutation survive in this case.

My thoughts on „Do you think that mutation testing is a feasible extension to unit testing?“

I think it’s quite a powerful tool to use. The mutation tests helped me to write more test cases which cover a bigger percentage of the code implementations.

Also, they helped me to encounter 2 of 4 found bugs.

But also 1 bug was not found by the initial test set and not by the mutation test, but by trying to think from different angles.

Fazit: Use mutation tests, but don’t rely fully on them!

Afterwards my mutation test looked like this:

Pit Test Coverage Report

Package Summary

at.ac.tuwien.inso.peso

Number of Classes	Line Coverage		Mutation Coverage	
2	100%	71/71	98%	46/47

Breakdown by Class

Name	Line Coverage		Mutation Coverage	
MobileMessage.java	100%	8/8	100%	2/2
MobileStorage.java	100%	63/63	98%	44/45