**Chapter 9**

Ex. 9.1 – Add several comments to the sales item while keeping an eye on the inspector for the comments list.

Done and works.

Ex 9.2 - Check that the showInfo method correctly prints the item information, including the comments.

Done and works

Ex 9.3 – Check the getNumber of comments method. Does it work correctly?

Yes, It works correctly.

Ex. 9.4 – Now check that duplicate authors are correctly handled.

Duplications are handled as expected

Ex. 9.5 – Perform a boundary checking on the rating value. (Test min, med, and max ratings)

Yes this works too.

Ex. 9.6 – Good boundary testing also involves testing values that lie just beyond the valid range of data. (test 0 & 6).

It was possible to leave a 0 star rating on the item. It can be fixed by changing the calidation line to “return rating < 1 || rating > 5;”

Ex. 9.7 – Test the upVoteComment and down vote comment methods. Make sure the votes are being properly balanced.

The up and down seems to work correctly with enough down it becomes negative and with enough up votes it becomes positive.

Ex. 9.8 – Use the upvote and down vote methods to mark some comments as more or less helpful then test the findMostHelpful comment method

.When comments are present it works as expected.

Ex. 9.9 – Do boundary testing on get mosthelpful. Does it work as expected?

If there is no comments in on an object it doesn’t work. A NoSuchElementException is given.

Ex. 9.10 – In the test exercises above you should have found two errors. After fixing them is it safe to assume that all previous tests will still work as before?

If fixed correctly everything should work as before. But whenever a change is made all features related to it should all be tested.

Ex. 9.11 – which of the test cases mentioned in the previous exercises are positive and which are negative test? Can you think of more negative tests to try?

From the previous exercises 9.1, 9.2, 9.3, 9.5, and 9.6 are all positive tests, 9.4, 9.6, and 9.9 are all negative tests, and 9.7 was a positive and a negative test. A couple more negative tests try are making sure the rating is the correct type (i.e. not a double 4.5 etc.). We could also test what happens if we try to remove and invalid index.

Ex. 9.12 – Run tests in your project using the Run Tests button. You should see a window similar to the one shown in figure 9.4.

Done

Ex. 9.13–Create a test class for the comment class in the online-shop-junit project.

Done and saved.

Ex. 9.14– What methods are automatically generated by the test class when it is created?

There are two methods currently in the test class and they are “setUp” and “teardown”.

Ex. 9.15– Create a test to check that addComment returns false when a comment from the same author already exists.

Done and saved.

Ex. 9.16 – Create a test that will perform a negative test on the boundaries of the ratings range. Test the values 0 and 6.

Done and Saved

Ex. 9.17 – Run all of the tests again. Explore how the dialog displays the failed test. Select the failed test in the list what options do you have available to further investigate the failure?

We can click on the error and we get the option to view the source. When we do this the line of code that caused the failure is highlighted making it far easier to see where mistakes were made.

Ex. 9.18 – Create a test class that has Comment as its reference class. Create a test that checks whether the author and the ratings details are stored correctly after creation. Record separate tests that check whether the upVote and downVote methods work as expected.

Done and saved.

Ex. 9.19– Create tests for SalesItem that test whether the findMostHelpfulComment method works as expected.

Done and Saved

Ex. 9.20– Add further automated tests to your project until you feel reasonably confident the classes are operating as expected.

Done and saved.

Ex. 9.21– Re-write all loops in the SalesItem class using streams. No explicit loops can remain and the program should not lose any functionality.

Done with one syntax error

Ex. 9.22– Make sure in the classes in the project compile. Then create a tester object and test all methods. What is printed in the terminal window? Do you believe the final line of what it says?

The object prints the following.

The terminal prints this

Testing the addition operation.

The result is: 7

Testing the subtraction operation.

The result is: 5

All tests passed.I do not believe the test because even after looking at the code it only checks a certain case so it can’t be said to test all possibilities. Perhaps it won’t work if something is becomes negative or something similar.

Ex. 9.23– Using the object you created in the previous exercise call the testPlus method. What result does it give? Is this the same result as printed by testAll? Call it again and what does it give? Should it always give the same answer? The first result given is 1 and that is different than what the testAll() method printed. If called a second and third time it displays 7. When looking at the code it should always give 7 as the answer.

Ex. 9.24– Repeat the previous exercise with the testMinus exercise.Does it always return the same result?

No, it returned 5 at first then after that it started returning -13.

Ex. 9.25– Perform a similar walkthrough of your own testMinus method. Does that raise any further questions about the calcEngine?

After going through it it seems to work on paper but to be sure I would like to chack the clear, numberPressed, minus, and equals methods. After seeing it I can determine with relative certainty that the methods will be successful.

Ex. 9.26– Complete the state table based on the following subsequent calls found in the testPlus method: numberPressed(4); , equals(); , getDisplayValue();

|  |  |  |  |
| --- | --- | --- | --- |
| Initial State | 0 | 0 | ‘ ‘ |
| Clear() | 0 | 0 | ‘ ‘ |
| numberPressed(3) | 3 | 0 | ‘ ‘ |
| Plus() beginning | 3 | 0 | ‘ ‘ |
| Plus() end | 0 | 3 | ‘+’ |
| numberPressed(4) | 4 | 3 | ‘+’ |
| Equals() | 7 | 0 | ‘+’ |
| getDisplayValue() | 7 | 0 | ‘+’ |

Ex. 9.27– When walking through the equals method did you feel the same reassurances that we felt in applyPreviousOperator about the default value in previousOperator?

No, I done’t feel the same. This is because of how it checks previous operator. It only checks to see if it is ‘+’ and if it isn’t it assumes ‘-‘. When only adding and subtracting this isn’t an issue but if the program was to be changed in the future we would want it to check explicitly for ‘+’ or ‘- ’ and print some kind of message if it is neither.

Ex. 9.28– Walkthrough the clear method immediately following a call to the getDisplayValue() method. Is the engine in the same state as it was at the previous call to clear? If not, what impact do you think this will have on any subsequent calculations?

The engine’s state changes once clear is called. The methods plus() and equals() make it so the three variables never return to their initial states again.

Ex. 9.29– In the light of your walkthrough, what changes do you think should be made to the calcEngine class? Make those changes to a paper version then walkthrough again.

Changed the clear and equals methods. Done and Saved.

Ex. 9.30– Try a walkthrough with the following sequence of calls. What should the result be? Does the engine

The answer should be 6 and the program returns 6 as expected.

Ex. 9.31– Open the calculator-engine-print and complete the addition of print statements to each method and the constructor.

Done and saved

Ex. 9.32– Create a CalcEngineTester in the project and run testAll method. Does the output results help you identify where the problem lies?

Yes, It helps confirm that the previous operator isn’t reset after an operation is comleted.

Ex. 9.33– Do you feel that the amount of output produced by the fully annotated CalcEngine is too little or too much or just right? (remove unecssacary comments)

Yes, I feel there is way too much in the way of comments. Changes made and saved

Ex. 9.34– What are the respective advantages and disadvantages of using manual walkthroughs or print statements for debugging?

Manual walkthroughs and print statement are great for helping us understand the code itself and can help use pinpoint the section of code that is having issues. however, it can be slow and most of the added comments should be removed once you are done testing.

Ex. 9.35– Using the calculator engine project, set a breakpoint in the first line of testPlus method in the CalcEngineTester class. Execute this method, when the debugger appears, walk through the code step by step. Experiment with both the step and step-into buttons.

Done

Ex. 9.36 – In practice you will find that Hacker’s attempt to program the CalcEngine is too full of errors to be worth trying to fix. Instead write your own version of the class form scratch.

Done and Saved.

Ex. 9.37 – Open your calculator project and add a better method for testing and add a unit test class to perform similar tests to the one’s in hacker’s test.

Done and Saved.

Ex. 9.38 – Open the bricks project. Test it. There are at least 4 errors in this project. See if you can find and fix them.

Done and saved.