**Written Assignment – 07**

Lewis University

Software Engineering

February 26, 2023

**9.2**

Unit testing is popular with the JavaScript development community since JavaScript is often used to develop web applications, and many JavaScript tests are unit-based.In some languages, unit testing is known as mock testing, mock object testing, or sometimes the name of the method used in unit testing, such as test case or test method. The term unit test, as used here, means testing a complete unit. That is, a single unit's functionality is tested at once. This implies that the unit being tested is a discrete or self-contained item. In other languages, unit testing is called subsystem testing (Ferrari et al., 2020). A subsystem is a self-contained and independent component of a more extensive application. It is only sometimes desirable to test a component in isolation since it can sometimes be helpful to test a component that is not directly related to the one being tested.Unit testing is also known as internal testing or black box testing. Black box testing refers to the testing of the software from the outside. Internal testing refers to the testing of software from the inside. In software testing, "Unit Testing" is used for "Testing at the Unit Level."

Feature testing also includes non-functional, typically automated, and functional testing. During functional testing, a tester manually explores and tests the software application's functionality. During this process, the tester identifies user interface (UI) errors. However, as the software application becomes more it can be challenging to identify UI errors manually. When UI errors are not found during functional testing, such errors go unnoticed by the tester and may be overlooked by the QA team (Ferrari et al., 2020). Feature testing can also include non-functional testing, which is typically manual and occurs after functional testing. During non-functional testing, the QA team typically attempts to detect new features, functionality, and requirements that have yet to be found during the functional test. For example, during the non-functional test, the QA team may check whether a feature is operational and determine whether a new one is operational. The non-functional test may include one or more "end-to-end" tests that verify the complete functional process (e.g., from receiving an order to processing that order and then fulfilling the order). End-to-end testing often involves many manual steps. For example, non-functional testing can be very time-consuming since testing an application may take several hours. It can sometimes take weeks, or even months, to complete a non-functional test on a single application.

**9.3**

**Negative test case:**

Create a document without any headings in the document and then use the automatic contents list feature and verify that all of the headings are NOT included in the list.

**Positive test case:**

Create a document with several headings and then use the automatic contents list feature and verify that the headings are included in the list.

**Boundary test case:**

Create a document with a single section that contains several bodies and headings. Use the automatic contents list feature to create a list of the body and headings in the document. Now do the same operation on the section of the document (Ferrari et al., 2020). Verify that the list only contains the body and headings in the created section.

**Performance test case:**

Create a document with several headings and body sections and populate the list for those parts using the automatic contents list feature.

**Usability test case:**

Use the automatic contents list feature to create a list of all headings and the auto body list feature to create a list of all body sections.

**Integration test case:**

Create a document with several headings and body sections. Use the automatic contents list feature to create a list of all headings and the auto body list feature to create a list of all body sections.

**9.4.**

**The following six end-to-end tests could be used to evaluate the system's characteristics.**

**Negative test case:**

While Emma is creating the group in her app, Jamie calls her, and she tells him that she is in the middle of it. After the call, she returns to the app and realizes no one else has created the group. She looks in the iLearn and notices that this is the first time anyone else has viewed or accepted her group invitation (Moe, 2019). The group does not appear in the teacher's iLearn.

**Positive test case:**

Emma creates the group in her app. She creates a message to share with Jamie using any of the app's methods. She shares the message with Jamie, and his message appears in her "Unread Messages" list in iLearn. Jamie looks at the "Group Invitation" message and notices that it is from Emma. He then accepts the group invitation by clicking on "View Invitation." He immediately discovers the group is empty because he is the only member (Moe, 2019). He then creates a topic, writes an email to the group using any of the app's methods, and sends the email.

**Boundary test case:**

The user who created Emma's group is also using iLearn. When Emma sends the group invitation message to Jamie, it appears in her "Unread Messages" list in iLearn. Jamie then asks his teacher to show him the "Group Invitation" message on iLearn, and Emma's message appears in the teacher's "Unread Messages" list in iLearn.

**Performance test case:**

Emma is creating the group and sending an invitation message to Jamie. After that, Jamie accepts the group invitation, but the group is not created on the server side (Moe, 2019). The app has to create the group on the server side after Jamie accepts the group invitation.

**Usability test case:**

After Jamie accepts the group invitation and it becomes active, his teacher sends a message to all the group members and tells them to click the accept link to the group invitation. Emma is not a group member and needs to receive a message from her teacher (Wasowski & Berger, 2023).

**Integration test case:**

Two or more users are in the same app and are in a conversation. Then, one of the users joins a group using the app's iLearn feature. The group can be created only when both the users in the app are connected Internet.

**9.4**

def sum\_list(numbers):

    """ Returns the sum of a list of integers."""

    total = 0

    for num in numbers:

        total += num

    return total

Finally, here is an illustration of how to create automated tests for this function using Python's built-in unit test framework:

Import unit test

class TestSumList(unittest.TestCase):

    def test\_sum\_list\_correct(self):

        self.assertEqual(sum\_list([1, 2, 3]), 6)

        self.assertEqual(sum\_list([-1, 5, 10]), 14)

        self.assertEqual(sum\_list([]), 0)

        self.assertEqual(sum\_list([0]), 0)

    def test\_sum\_list\_incorrect(self):

        with self.assertRaises(TypeError):

            sum\_list("hello")

        with self.assertRaises(TypeError):

            sum\_list([1, 2, "three"])

Two test methods—test sum list correct and test\_sum\_list incorrect—have been defined in this example. When using valid data, the test\_sum\_list correct method checks the function; the incorrect method is used when using erroneous data.

The assert Equal method is employed in test\_sum\_list correctly to confirm that the sum of the integers in different lists is accurate. Also, test the extremes of an empty list and a list containing only one value of zero.

Utilize the assertRaises method in test\_sum\_list incorrectly to confirm that the function issues a TypeError when given erroneous data (in this case, a string and a list with a string value).

**9.8**

Refactoring can also be used to simplify the system by removing duplication. There are many refactoring patterns to perform, including these six:

* Extract method
* Extract Class
* Encapsulate field
* Fix method signature
* Move class
* Encapsulate field
* Renaming Class

Refactoring can be used to make scalability changes or improve code readability. With every refactoring, there is the risk of breaking existing functionality, so this technique is best applied after most of the code is completed. Moreover, refactoring should not be done when the code is pressured to deliver a new version when it is not possible to make enough tests, when there is not a good plan to make tests, or when the refactoring is only partially done. Refactoring is a change of the code without changing the external behaviour (Wasowski & Berger, 2023). It can be done during the maintenance process in several areas, like bug fixing, or when a class or method must be duplicated in several parts of the code. Refactoring is a part of the continuous integration process, where code can be modified and checked for possible errors before merging code to production environments. It can be done manually, in automated refactoring, or using formal techniques. The idea is to prevent or limit the introduction of harmful practices, like the violation of object-oriented design principles. One of the main points is that more than refactoring is needed to maintain quality in an existing system. Refactoring alone cannot prevent the accumulation of technical debt incurred due to improper design. However, refactoring can help to prevent these bad design practices. The first step of refactoring, though, is to ensure that all the code is developed correctly and following the team's coding standards. The code should be tested at the end of each module and the newly created or refactored one (Wasowski & Berger, 2023)

. Once all the tests are executed and passed, the developer can safely refactor the code. The refactoring should be done to optimize the code, remove duplication and improve the design.

**9.10**

Testing and code reviews are crucial steps in the software development process, and they work in tandem to raise the caliber of the products being created. The following three arguments support the usage of code reviews in addition to testing:

Raising quality:

Code reviews help improve software quality, directly resulting in higher-quality software. This is because code reviews allow a developer to provide valuable feedback about potential issues, and these issues can be fixed before a product is released.

Encouraging consistency:

The idea that each team member must produce code that adheres to specific standards and guidelines is called 'coding standardization.' Code reviews can help enforce this. Consistency is ensured throughout the software rather than relying solely on individual developers.

Increasing code quality:

Code reviews help identify and fix errors that may have previously gone unnoticed and help avoid issues before they occur. Software tests can also be performed similarly, but code reviews offer some advantages (Wasowski & Berger, 2023). One of the essential differences between code reviews and other forms of testing is the ability to detect problems before they occur, meaning that a code review can provide the added advantage of being able to help identify coding mistakes before a problem is encountered.

References

Ferrari, D., Motta, A., Strollo, M., Banfi, G., & Locatelli, M. (2020). Routine blood tests as a potential diagnostic tool for COVID-19. *Clinical chemistry and laboratory medicine (CCLM)*, *58*(7), 1095-1099.

Moe, M. M. (2019). Unit test using test-driven development approach to support reusability. *International Journal of Trend in Scientific Research and Development (IJTSRD)*, *3*(3).

Wąsowski, A., & Berger, T. (2023). Internal Domain-Specific Languages. In *Domain-Specific Languages: Effective Modeling, Automation, and Reuse* (pp. 357-394). Cham: Springer International Publishing.