**Mobile Junit5 Test Summary & Reflection**

Fellipe Demoraes

CS-320-H7332: Software Test Automation & QA

Professor Wilson

December 11, 2022

Mobile Junit5 Test Summary & Reflection

My task as a software engineer was to build out a mobile application. Currently, the project is on a unit level. In other words, the components that are supposed to work together in a software application is built, but an integration planning, development, and testing is needed as of right now. The project has also taken 3 weeks to complete. The software development lifecycle (SDLC) has re-cycled for each of the components, so development and testing for the component has been done first. This SDLC style has reduced the work testing is done at other test levels like system testing for example, and it also ensures that more time is assigned to testing, and it doesn’t tempt the project manager from taking time away from testing and prioritize other areas of the project, because testing is being done adjacent to the component’s development.

A week included reviewing the requirement documentation, planning how to implement the requirements, coding, and then testing. For ContactService, one of the requirements were that contacts had to be uniquely stored. I solved this requirement by giving ContactService a HashMap list, that stored the contact’s id’s as a key and stored the contact object itself as a value. Another benefit to using a HashMap was its processing speed. It doesn’t matter the list size due to it’s big o notation being O(1). Other requirements included: adding new contacts, deleting existing contacts, and updating a contact. These requirements were given these exact functionalities in ContactService. Testing was done for each of the functionality. In addContact, a potential bug would be adding duplicates into the list, so I tested that an IllegalArgumentException is thrown if a duplicate contact is added. For deleteContact and updateContact, the functionality requires that a contact exists first before its executed. Both functionalities are tested for an error if the contact doesn’t exist. The last requirement was for no null values. With Java, the language itself will throw a NullPointerException error if an object operation is done. This has caused me comment out scenarios where null values exists in the Contact class, but this error along with the IllegalArgumentException will have to be handled during an integration phase. A best practice would be checking for null values first, and then throwing error messages that aid other developers during debugging. This best practice was applied though during the Task, and Appointment development and testing phase. The TaskService has the same functionalities as the ContactService. The only difference with AppointmentService is that only 2 functionalities are required and implemented: addAppointment and deleteAppointment. In terms of testing quality, every component created was tested as it was developed, and testing were specific to the requirement specification. Thus, if there is a requirement, then it’s tested. Additionally, even though the methods, like deleteAppointment created for the requirements specified didn’t require any return value, a Boolean return value was added to ensure two things: that the operation was successful, and that the code reached 100% test coverage. On lines 57-61, the deleteAppTest is testing that the return value is true with assertTrue, and this was added specifically for test coverage purpose. The problem was that null, and non-existing objects were being checked, so the program would never continue to remove the object itself. This was also not possible to test because of a no return value. Similar test coverage approaches were done in ContactService deleteContact, updateContact, and TaskService deleteTask, and updateTask.

Testing techniques used in the project so far are equivalence partitioning, statement and coverage tests, decision testing and coverage, discerning the value of the coverage, and an experience-based testing technique as well. Equivalence partitioning was utilized when checking the string field length, so one value was used to test if the field length was greater than the condition specified. Regarding test statements, the goal was to meet the rules and conditions and to keep it simple. If required, more functionality and testing can be added during the integration testing phase, which I am expecting to do. For test coverage, not all were 100%, but this is due to a low priority value. For example, it doesn’t help to test getter methods when it’s returning an expected value. This of course isn’t the case for every project depending on the complexity. Another observation is line 19 coverage of AppointmentService class is not reached. This would not work with a assertTrue assertion because the class itself doesn’t have any functionality that would allow me to test if the correct object was deleted. Instead, this problem can be fixed in terms of coverage by adding assumeTrue assertion. However, I am expecting that more functionalities will be built during the integration development and testing phase, and the delete method functionality will be tested to make sure that the correct object is removed. This is in line with the software development lifecycle best practices, where the code is tested as it gets built.

Testing techniques not used were for instance the use of test suites to run all the test classes at once. This would be beneficial if there were many test classes to go over in a project. This can be the case during system testing phase, where it’s important that the system and all of its components are working as expected. Boundary value analysis wasn’t used, and this is because the properties were string and date values. If the project was high risk and decimal values were used, then this would be considered a high-priority test. Other techniques like decision table testing, state transitioning testing, and use case testing aren’t used because the project is not at the level yet. The project thus far is at a unit test level, so not all test techniques apply. The other levels are integration testing, system testing, and acceptance testing. At the system testing stage, the components built are integrated and work together; the idea is to make sure that it works as expected. A decision table testing technique would be applied in this case; it’s a specification that contains business rules that make up the system’s functionality and conditions, and this is what the tests are checking.

My mindset during the project was test as you go, as this is considered a SDLC best practice for good reason. In the waterfall method, development of a system would occur, and a test of the system would occur after. This has potential implications because now the software developers in charge of testing the system must perform unit testing, integration testing, system testing, and acceptance testing. Also, it may not be every project that prioritizes testing as much because deadlines and budgets. In short, the quality of testing isn’t as great and that results in an application that may never be launched to production or if it does, it will cost the company a large amount of money to maintain. Additionally, I made it important to not overcomplicate the project and stick to what is being required in the software. This relates to adding a Boolean return value during test coverage. I knew that it would be useful to return a value during these operations, but I had no idea how it would be used, so I left the return value void until I had a use for it during test coverage analysis. I also had in mind having consistency in my program. For example, if ContactService and TaskService had similar functionalities, then I should try being consistent with both. If we look at both their constructors, null and length checks are being performed and both classes have getter methods. This allows a developer to understand the code at a faster rate than having to spend additionally time understanding how each class works. On the issues of developers testing their own code, it’s important for developers to test it because no one knows their code better than they do at the time. A good software developer should know the importance of testing their code as well because every developer has that experience trying to debug their own software, and not effectively testing their code could result in system failure that could potential cost millions of dollars or lives. In addition to developers testing their own code, software should be tested by another developer as well depending on the level of risk the software poses. This helps ensure that no mistakes were made. A test document should be created showing what tests were done and it should be provided to the next software developer. The next software developer can easily determine what additional tests should be performed.

Testing is an important aspect of SDLC. Without it, system malfunctions would occur that potentially causes millions of dollars of damage, or it causes loss of life. Take for instance the Ariane 5 rocket explosion. The project was years in the making and it costed at least $500 million dollars. The cause was a floating-point value conversion to 16 bits signed int, and it wasn’t big enough to contain the values assigned to it, causing an error. What about vehicles? Cars are becoming smarter and becoming more dependent on technology. Toyota had to recall 460,000 vehicles over a software error. Depending on how the vehicle was turned on, the Vehicle Stability Control system was turned off. This caused the vehicles to no longer follow regulation, so the company was required to make this fix. This would have potentially been avoided with 1 unit test and saved the company trouble and cost. Due to the importance of testing, this should be in mind of more than just developers. Even if it doesn’t relate to regulation or risk of loss of life, errors cost a lot of money to fix during production as per the cost escalation model. Here are the comparative costs relative to when the error is found: 1$ in requirements, $10 in Coding, $100 in program testing, $1,000 in system testing, $10,000 in user acceptance testing, $100,000 in live running.

References

Arnold, D. (2000). *The Explosion of the Ariane 5.* <https://www->

users.cse.umn.edu/~arnold/disasters/ariane.html

Demoraes, F. (2022). *Software-Testing*. GitHub. https://github.com/fddemora/software-testing

*Detail the big-O of Hashmap - put() method by real code in Java 8*. (2019b, March 20). Stack

Overflow. <https://stackoverflow.com/questions/55263115/detail-the-big-o-of-hashmap-put->

method-by-real-code-in-java-8

*Toyota Recalls 460,000 Vehicles Over Software Error.* (2022)International Business Times.

<https://bi-gale-com.ezproxy.snhu.edu/global/article/GALE%7CA700431076?u=nhc>

\_main&sid=ebsco