Ryne Williams

Southern New Hampshire University

CS 320 7-2 Project Two

Summary:

A: Testing Approach

When developing the tests for the software in development I created tests that would check the functionality of each component individually. I set up a test specifically to test the add methods for each part of the program to ensure that the application successfully created and saved the new item (task, contact, or appointment), as well as ensuring that the proper number of contacts remained in the ArrayList upon successfully saving the new item. I also made sure to test that only a unique ID number could be used when creating a new list item for each part of the program. I also extensively tested that list items were able to be modified for the sections of the program that required the ability to modify items. It tested that each individual part of the ArrayList could be modified without affecting the other items in the ArrayList, as well as ensuring that the modified items were within the length requirements of the program. I tested the “delete” functions of all three separate parts of the software to ensure that when a list item was deleted, it reflected in the ArrayList count, as well as not being able to be found when calling the specific ID number for the deleted item. Finally, I made tests to ensure that all items received as user input met the requirements regarding length and nullity, ensuring that all items had valid inputs. The quality of the JUnit tests created for the classes containing the getters and setters was high, with overall above 80% coverage. Also, the coverage for the add, modify, and delete methods was at a minimum of 89.7%, leading to the confidence of high quality in those methods for each class in the program. However, due to oversight and some slight confusion, the main methods that call the add, modify, and delete methods were not tested. An oversight that will be fixed and have more focus on in the future.

B: Testing Experience

Using proper annotations and assertions with my JUnit tests, I was able to ensure that the code in the program was technically sound. I used annotations to create setups for each test in each test class. Using “@BeforeEach” I was able to create the same testing standard for each test case to ensure that no other test affected the outcome of another test. Doing this ensured that the proper functionality of each test was properly checked by the JUnit tests, and all tests were performed against the same standard. The use of assertions ensured that the desired output of the program would be what was expected based on the requirements given and no unexpected errors occurred due to unexpected input. Finally, the use of “@DisplayName” allowed the tests to be more easily understood as to what their purpose was. Allowing others who view the tests to know where to look in order to check those tests.

*@BeforeEach*

void setUp() {

ContactService.*contactList*.clear();

}

*@Test*

*@DisplayName*("Tests for addContact method")

void addContactTest() {

//Test that addContact adds contact with unique ID

ContactService.*addContact*("14532", "John", "Smith", "5557894321", "55 Madison Blvd");

Assertions.*assertEquals*(1, ContactService.*contactList*.size());

//Test that addContact does not allow non-unique ID

Assertions.*assertThrows*(IllegalArgumentException.class, () -> {

ContactService.*addContact*("14532", "Amanda", "Sanders", "5558315645", "715 Capshaw Road");

});

}

By creating individual test cases to test against each individual component of the software I was able to ensure that the code was efficient and operated properly. Testing each component individually allows for the tests to check each part against most unexpected inputs and can check for unexpected errors or outputs. This also will allow testers to be able to trace any issues or errors back to the source of the bug and fix it more quickly. Furthermore, individual tests can help to ensure that the proper inputs will give the proper outcomes.

*@Test*

*@DisplayName*("Tests for Appointment ID")

void testAppointmentIdTooLong() throws ParseException {

Date date = new SimpleDateFormat("dd/MM/yyyy").parse("16/08/2023");

Assertions.*assertThrows*(IllegalArgumentException.class, () -> {

new Appointment("68419735085", date, "Birthday Party");

});

}

*@Test*

void testAppointmentIdNull() throws ParseException {

Date date = new SimpleDateFormat("dd/MM/yyyy").parse("16/08/2023");

Assertions.*assertThrows*(IllegalArgumentException.class, () -> {

new Appointment(null, date, "Birthday Party");

});

}

*@Test*

*@DisplayName*("Tests for Appointment Date")

void testAppointmentDatePast() throws ParseException {

Date date = new SimpleDateFormat("dd/MM/yyyy").parse("16/08/1990");

Assertions.*assertThrows*(IllegalArgumentException.class, () -> {

new Appointment("8675309", date, "Birthday Party");

});

}

*@Test*

void testAppointmentDateNull() {

Assertions.*assertThrows*(IllegalArgumentException.class, () -> {

new Appointment("8675309", null, "Birthday Party");

});

}

Reflection:

A: Testing Techniques

In creating tests for Project Two there were several testing techniques that were employed to ensure that the code being developed was properly tested. These tests were developed using the Static Testing, White Box Testing, and Unit Testing techniques. Static testing is a testing technique that checks the code for proper functionality before the code is ever executed, making it a very quick test with no need to know how the program is supposed to run. White Box Testing is a technique that goes hand in hand with static testing, as it is a technique that is done without the need for a user interface, it is performed solely on the code itself. Finally, unit testing is employed by testing each component of the program individually to ensure that they each function properly on their own.

The testing techniques that I did not use in Project Two include Black Box Testing, Regression Testing, and Exploratory Testing. The scope of the project made it unfeasible to begin using these testing techniques. Black Box testing is the counter part to White Box testing, in which the tests are centered around the user interface and how the program reacts to user input. This is done with little to no knowledge of the inner workings of the program and its code. Regression testing is performed on code that has been previously tested and then returned to the testers to ensure that the fixes were accomplished successfully and did not adversely affect any other part of the overall program. Exploratory testing is done with little planning and knowledge of the code and its purpose. It is a very effective testing technique if flexibility is needed, and testers may need to change the path in which they test suddenly.

When in the beginning stages of the Software Development Lifecycle, it is often not possible to test code since the user interface has not been created yet. This is where Static testing would be most useful, as it is executed without ever having to execute the code. Similarly, White Box testing is most often used at the beginning of software development. The documentation is checked to ensure that it matches the client’s requirements, and the code is tested without the need for a user interface, mostly by ensuring that the code matches the requirements. This is also the best time to perform Unit testing, while the components have not been integrated with other parts of the application. This way, testers can ensure that each individual component will more likely interact with the overall system in the way it was intended. Regression testing, however, is used later in the development process. This technique is used to retest code as it has been fixed from previous testing to ensure that it has been properly fixed and that it will not adversely affect the other components of the system that have moved on to the implementation process. Black Box testing comes shortly after, where testers will use a user interface to ensure that the entire system will operate as intended and to client requirements. This is done with little to no knowledge of the coding and backend work that has been done. Finally, Exploratory testing is performed when problems are known, but there is little knowledge of the system or source code. This is done with little to no planning but allows testers to change their testing strategy quickly in order to better test the system under test as they perform their tests.

B: Mindset

When creating tests for the software being developed it is important to exercise caution in the tests being created. Caution can allow the tester to reassess the tests being created to ensure that they are not only effective, but also relevant to the requirements that the client has set. Creating tests for software components is only useful if it is testing what the software is supposed to do, and not arbitrary assumptions that the tester may think are relevant. It is also important to consider how complexity can change the tests that need to be created in order to properly test the code. The more complex the code, the larger it will be. This will create a need for more complex test cases to ensure that all aspects of the code have been properly tested. Also, with more complexity, comes with more concern for how the code will interact with other components of the software and the level of care that must be taken is much higher than if the code were simpler.

It is often the case that when a software developer is testing their own code that there may be some bias in how they would test their own code. For example, it is likely that they may have a high level of confidence in their coding ability and feel as if there is little to no need to test their code. While their skill level may be high, humans are always prone to mistakes, and as such need to be checked to avoid that no mistakes make it into the final product. This is a good cause for concern in the bias that a developer may have in their own code when considering quality. I personally have a desire to see that anything that I have a hand in creating be fully functional in a way that makes it as useful as possible. Therefore, I want to test any code that I have made as extensively as possible to ensure that I am not handing off buggy code that is full of mistakes that could have easily been fixed had proper testing been performed.

A good way to ensure that the developer is not the reason that a project goes over budget due to late-stage testing and repairs to code is to be committed to ensuring the quality of the code being developed. This will not only help with the budget, but also the projects timeline and the developer’s credibility. When a software tester ensures that all code is of the highest quality that can be reasonably attained, then the Software Development Lifecycle will run more smoothly, and the process will progress more rapidly. I intend to employ the ideology of “test early and test often”. This will be to ensure that any code that I create is up to quality standards and meets requirements, if I am the one testing it. If I am testing code developed by other software engineers, then developing extensive and exhaustive test cases against reasonable unintended user input would help to ensure that any defects, bugs, or errors in the code that I am testing has been found and that they can be fixed before moving on to later stages in the Software Development Lifecycle.

References

Hambling, B., Morgan, P., Samaroo, A., Thompson, G., & Williams, P. (2019), *Software testing: An ISTQB-BCS certified tester foundation guide - 4th edition*, BCS Learning & Development Limited, [https://ebookcentral-proquest-com.ezproxy.snhu.edu/lib/snhu-ebooks/detail.action?docID=5837074#](https://ebookcentral-proquest-com.ezproxy.snhu.edu/lib/snhu-ebooks/detail.action?docID=5837074)

Garcia, B., (2017), *Mastering Software Testing with JUnit 5: A Comprehensive, Hands-on Guide on Unit Testing Framework for Java Programming Language*, Packt Publishing, <https://eds-p-ebscohost-com.ezproxy.snhu.edu/eds/detail/detail?vid=0&sid=4394f12c-8a70-4f02-a07e-579a64e54f4a%40redis&bdata=JnNpdGU9ZWRzLWxpdmUmc2NvcGU9c2l0ZQ%3d%3d#AN=1626950&db=nlebk>

Boog, J., (ND), 9 Types of Software Testing in Software Engineering, QA Lead, https://theqalead.com/test-management/types-of-software-testing/