Jonathan Miller

Prof. Kalysa Wilson

CS-320

August 13, 2023

Project 2

Summary

Testing Approach

- *To what extent was your testing approach aligned to the software requirements?*

My testing approach for the project involved assessing the requirements, implementing the software based on the requirements, and then the different units that aligned with the requirements were individually tested. Each feature, derived from the requirements, corresponded to one or more units that needed to be tested.

For example, the requirements needed each service to add, delete, and update the contacts, tasks, and appointments. So, each of these requirements corresponded to a method in the service classes. Therefore, the contact, task, and appointment systems were tested using white-box structural testing that closely aligned with the requirements.

- *How do you know that your JUnit tests were effective on the basis of coverage percentage?*

To determine the effectiveness of my JUnit tests, I used the white-box testing technique of coverage percentage. (Garcia, 2017, pg. 284) I tried to make sure that the JUnit classes covered a majority of the corresponding class that they were testing.

For example, the *TaskService* class had a corresponding *TaskServiceTest* class. After running the *TaskServiceTest* class as a coverage test, I would then check the test's coverage of the *TaskService* class. If it was around 80% or above, this would be considered acceptable (Although, I strived for 100% test coverage because of the smaller scope of the project.)

I was able to get the *ContactTest* to cover 100% of the *Contact* class, and the *ContactServiceTest* covered 100% of the *ContactService* class. I was also able to get the *TaskTest* to cover 100% of the *Task* class, and the *TaskServiceTest* covered 100% of the *TaskService* class. And lastly, I was able to get the *AppointmentTest* class to cover 100% of the *Appointment* class, and the *AppointmentServiceTest* class covered 100% of the *AppointmentService* class. Therefore, these test classes had an excellent coverage rate, and they were able to thoroughly test the system.

JUnit Testing Experience

- *How did you ensure that your code was technically sound?*

I was able to ensure that my code was technically sound by making sure that my tests were simple, easy to read, and only had a single responsibility. I also tried to avoid anti-patterns such as *piggybacking* and code smells such as highly complex, too large, or inappropriately named test cases. (Garcia, 2017, pg. 288-290) My test cases also did not depend on other test cases so that they could be ran independently of each other in any order. I used assertions to make sure that each test produced that intended result.

For example, here is my test case to ensure that a too long task description produced an exception:

```
@Test
void testTooLongDescription() {
	// Verify an exception was thrown for too long description.
	Assertions.assertThrows(IllegalArgumentException.class, () -> {
		new Task("1", "Task Name", "*".repeat(51));
	});
}
```

In this test case, I had a descriptive function name that effectively describes what the test is testing. The body of the test function is concise and easy to read. I used the "assertThrows" function to prove that an IllegalArgumentException was thrown when creating a new Task with a description that had one too many characters. To simplify the creation of the too long description, I used the Java string function "repeat" to concisely create a string of 51 asterisk characters. This made the test case much more readable than manually typing out a description that had a length of 51 characters, and it also ensured that the description consisted of exactly 51 characters.

- *How did you ensure that your code was efficient?*

To ensure that my code was efficient, I made sure that my test cases were not overly complex or too long. I also did not use any time-consuming methods in my test's cases. I used the JUnit window to ensure that my test cases took under a second to complete, and after analyzing the test results, they did.

For example, to test if a Task was correctly added to the tasks hash map, I only needed to use the "containsKey" method provided by the built-in Java HashMap class. Here is the code for that test case:

```
@Test
void testAddTask() {
        // Test object.
        TaskService taskService = new TaskService();

        // Try to add a valid task.
        taskService.addTask("1", "Task Name", "Task Description");

        // Verify the task was added.
        assertTrue(taskService.getTasks().containsKey("1"));
}
```

I chose a data structure that was efficient at searching for specific values so that I would not have to iterate through a container until the key value was found. The “containsKey” method has a search complexity of O(1), which is very efficient. My “getTasks” method simply returns the tasks hash map, and then chaining was used to check if the hash map contained the task with the specified task ID. The usage of the HashMap “containsKey” method and chaining in the assertion made checking for an added task easy, concise, and efficient. I chose the HashMap data structure to contain the objects in each service class due to its fast, constant access time. (Oracle, 2023)

Reflection

Testing Techniques

- *What were the software testing techniques that you employed in this project?*

The project required dynamic testing techniques that involved unit testing. This means that each test was performed on a specific unit of the System Under Test. The tests were dynamic because the SUT code needed to be running to perform the unit tests.

White-box testing on the internal structure and implementation was performed on each SUT. Each internal method translated to a small set of unit tests. For example, one requirement was that the *AppointmentService* class needed to provide a means to add a new appointment.

This requirement translated to an *addAppointment* method in the *AppointmentService* class that needed a set of three different unit test cases – the *testAddAppointment* unit test case, the *testAddInvalidAppointment* unit test case, and the *testAddDuplicateAppointment* unit test case. These three test cases resulted in a high test coverage of the *addAppointment* method because the entire structure and functionality of the method was tested.

A test-last development methodology was utilized for the testing of this project. Each SUT was designed and implemented before the test cases were designed and implemented. As hinted at already, testing was performed on the basis of test coverage. Therefore, the test cases were designed to cover as much of the internal structure and functionality of the SUT as possible. Since the project required a rather small implementation, it was possible to aim for 100 percent test coverage for each SUT.

- *What are the other software testing techniques that you did not use for this project?*

Software techniques that were not utilized in this project included black-box testing techniques such as equivalence partitioning, boundary value analysis, decision table testing, state transition testing, and use case testing. (Hambling et al., 2019, pg. 101-102) I also did not perform static analysis using external tools, various forms of non-functional testing, and test automation.

Black-box testing is testing on the basis of requirements without any knowledge of the underlying structure, implementation, or system architecture. Equivalence partitioning is a black-box testing technique to determine how many tests are needed based on a partitioning of a domain of possible valid and invalid inputs. Representative values are utilized for each domain of possible inputs to reduce the number of test cases. Boundary value testing can be utilized to test the boundaries of each equivalence class. Decision table testing can be used to test the conditions that are required to produce specific action for a business requirement. State transition testing is used to test the behavior of a system when it transitions states. And lastly, Use-case testing can be used to test process flows and the behavior of scenarios on the business process and/or system levels. (Hambling et al., 2019, pg. 101-114)

Static testing is a technique that tests the SUT without actually executing the code. This type of testing can be performed through formal and informal reviews of the code. Static analysis can be performed through the usage of tools that analyze different aspects of the quality of the code. (Garcia, 2017, pg. 308-311) Then we have non-functional testing which is used to test the non-functional aspects of the SUT. These non-functional aspects include the performance, usability, reliability, and security of the SUT.

Test-driven development is a methodology of testing in which test cases are designed and implemented before the implementation of the SUT. The tests will be designed based on the analysis of the requirements. (Garcia, 2017, pg. 293)

In continuous integration software development environments, test automation will be performed. This automation is the automatic running of a set of defined tests during the integration of new code into the system. (Garcia, 2017, pg. 294-296)

- *For each of the techniques you discussed, explain the practical uses and implications for different software development projects and situations.*

Black-box testing consists of equivalence partitioning, error guessing, boundary value analysis, decision table testing, and state transition testing that tests the requirements and functionality of the SUT. White-box testing consists of statement coverage testing, branch coverage testing, and path coverage testing to test the structure and implementation of the SUT. (Hambling et al., 2019, pg. 115)

Black-box testing is a technique that is utilized by testers that do not know the underlying system architecture, structure, and implementation. Typically, black-box techniques are utilized by system testers and acceptance testers. Black-box testing does not require programming knowledge. (Hambling et al., 2019, pg. 101)

White-box testing, on the other hand, requires programming knowledge and access to the code base. In white-box testing the system architecture, structure, and implementation of the SUT is tested. Unit testing, algorithm testing, and integration testing utilize white-box testing techniques. (Hambling et al., 2019, pg. 115 - 144)

Mindset

- *In acting as a software tester, to what extent did you employ caution? Why was it important to appreciate the complexity and interrelationships of the code you were testing?*

Caution is employed regularly as a software tester. In practice, it is not always possible to achieve 100% test coverage for the SUT. Real-world software systems are complex with many relationships among many different units, and it would be nearly impossible to cover every single outcome in the system. It is only really possible to achieve 100% test coverage in software systems that are smaller in scope.

Therefore, risk management is necessary to determine what tests would provide the most benefit to the system. The tester should plan and define their tests to minimize the potential of a defect occurring in the live environment. (Hambling et al., 2019, pg. 158)

For example, a tester may have decided that the database retrieval mechanisms for a software are considered high-risk because it could result in the leakage of private personal information. So, they spent more time and resources on testing the security of the functions surrounding retrieving information from the database to ensure minimal defects regarding this aspect of the system. The testers decided to spend more time and resources on the former aspect of the system rather than on changing the theme of the interface because there was more risk involved in the database retrieval mechanisms.

- *On the software developer side, can you imagine that bias would be a concern if you were responsible for testing your own code?*

There is a bias concerning the correctness of the code and documents that were created by the developers. Software developers are less likely to see their own errors because they are the creator and owner of the documents and code that they have developed. Software developers perceive deliverables as being correct when they are delivered, and thus creating a bias that they have developed correct code or documents. Therefore, independent testing can be performed on the deliverables to eliminate this bias because the independent testers will assume that the deliverables are likely to contain errors. (Hambling et al., 2019, pg. 162)

For example, a developer may implement an update operation for a service class and assume that it was correct once it was delivered. However, the developer did not update the correct field in the update operation. Once the independent tester received the service class deliverable, they created a unit test case to test that the field was updated correctly, and they had discovered that the method did not update the field that it was supposed to. This is an example of how independent testing can overcome the bias that occurs when testing your own code.

- *Why is it important not to cut corners when it comes to writing or testing code? How do you plan to avoid technical debt as a practitioner in the field?*

  It is very important to not cut corners during the development process. Although, sometimes it is necessary in order to stay within a set of constraints such as time and budget. The topic of technical debt correlates directly with the topic of risk management, proper planning, and correct implementation.

  The risks that are taken in a software development project should be those that have the least costly impact, and this will lead to less technical debt. Proper assessment of the project and product risks will result in the least amount of technical debt. Also, proper development and test planning will also contribute to less wasted time and resources, resulting in less technical debt. Developers should also strive to produce correct code to relieve the testers of all the burden for finding defects since it is impossible to catch all of them. The less defects that are in the system, the less code correction and refactoring will need to be done, resulting in less technical debt. Therefore, proper risk management along with fewer sacrifices in system planning, design, implementation, and testing will result in the least amount of technical debt.

Cutting corners in software development could lead to significant costs such as a loss of revenue and reputation, but cutting corners in software development could also harm the wellbeing of people. For example, the inertial referencing system of the Ariane 5 space launcher had an overlooked defect regarding type conversion that led to an exception code being sent to an on-board computer that was interpreted as navigational data. This software defect led to the maiden flight of Ariane 5 ending abruptly as it self-destructed roughly 40 seconds after initiation of the flight sequence. The failure of Ariane 5's maiden flight cost the ESA hundreds of millions of dollars, and it hurt their reputation. (Le Lann, 1996)

References

Oracle. (2023). Class HashMap<K,V>. HashMap (Java Platform SE 8). https://docs.oracle.com/javase/8/docs/api/java/util/HashMap.html

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.

Garcia, B. (2017). *Mastering Software Testing with JUnit 5 : A Comprehensive, Hands-on Guide on Unit Testing Framework for Java Programming Language*. Packt Publishing.

Le Lann, Gerard (December 1996) The Ariane 5 Flight 501 Failure - A Case Study in System Engineering for Computing Systems. (Research Report) RR-3079, INRIA. ffinria-00073613f