**7-2 Project Two**

David J Allen

CS320: Software Test Automation & QA

Southern New Hampshire University

Dr. Lewis

April 16, 2023

**Summary**

**Aligned to software requirements**

The testing approach used was Junit tests designed around specificity requirements regarding string length and presence of null strings. The material presented requirements for a class and service for contact, task, and appointment. The contact object needed a unique ID that could not exceed 10 characters. In addition, the contact ID could not be empty or able to be changed. As part of the contact object, it also needed a first name and last name field no longer than 10 characters. A phone number field had to be exactly 10 digits and could not be empty. To round out the contact class there was a required address field that could not exceed 30 characters. All these requirements were used to incorporate the strings for building the class. The contact service utilized these created fields to be able to add contacts with a unique ID, delete contacts based on their ID, and modify the contact fields using the ID. The task and appointment objects only differed slightly in required strings and limitations. The task class - while needing a name, it was not first name/last name, it was an identifier for the specific task. The address and phone number strings became unnecessary and were replaced with a description field that could not be longer than 50 characters. For appointments the ID string requirements remained the same, but it added the newer element of data which could not be in the past and continued requiring the description field introduced in the task class. The service classes, just like contact, made sure to allow addition, deleting, and modification. An example from the code showing the declaration and requirement designation can be found below:

//Declaration of int/string for class

final public int ID\_LENGTH;

final public int DESCRIPTION\_LENGTH;

final public String DEFAULT;

public String appointmentId;

public Date appointmentDate;

public String description;

// Set maximum length for id and description

// Create default for data

{

ID\_LENGTH = 10;

DESCRIPTION\_LENGTH = 50;

DEFAULT = "Default";

Following through with the software requirements revolved around setting strings, max characters, and aptly named test cases. “Adhering to a properly-defined methodology enables a project to provide better estimates, deliver stable systems, keep the customer informed, create a clear understanding of the task ahead, and identify pitfalls earlier, allowing for ample time to make adjustments” (Segue Technologies, 2015, para. 3).

**Overall quality of your JUnit tests**

The coverage quality of the Junit tests was specified in the aptly named tab in Visual Studio where percentage was above 80% across all test criteria. While the test percentage number is not indicative of 100% coverage, the tests were built solely around the requirements listed and to throw illegal argument exceptions. “For all experiments, it is vital that inclusion generally examines the entire code. This is a conclusive and factual verification that all testable code is without a doubt tried” (Educba, n.d., para. 4). The first tests for ID involved a length too long and null to make sure the exceptions were thrown and passed. The name tests involved a length too long and null, again passing completely. The description test involved a length too long and null to make sure the exception was thrown. Date tests involved checking dates from the past against set date. The coverage test percentages can be seen in the screenshot below:

Graphical user interface

Description automatically generated

**Technically Sound and Efficient**

To ensure the code was technically sound every block of code contained specific comments to the functionality of the arguments. The code also made sure to use if/else statements to efficiently check the condition and throw the exception “Not correct” if it did not meet requirements. The tests were also efficiently named for what they were looking to accomplish for length, empty, and add or delete such as testAddAppointment and testContactIdLength. These tests required less commenting due to the obviousness of their purpose. Test strings were used that contradicted the requirements to ensure correct usage and coverage percentages. An example from the code that used the naming conventions can be seen below:

// Length and null test for contact id

@Test

void testContactIdLength() {

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

Contact contact = new Contact("CS320000000", "David", "Allen", "5181234567", "123 Center Way");

});

} //close id length test

@Test

void testIdNull() Assertions.assertThrows(IllegalArgumentException.class, () -> {

Contact contact = new Contact(null, "David", "Allen", "5181234567", "123 Center Way");

});

} //close id null test

**Reflection**

**Testing techniques used**

The software techniques that I used in the three milestones consisted of many types. The assignments encompassed functional, non-functional, white-box, testing related to changes, and dynamic testing. Functional testing checks on “the specific functionality of the system and is carried out at all levels of testing” (Hambling et al., 2019, p. 66). This can be seen through the JUnit tests run to make sure that the requirements functioned as intended. Non-functional testing looks for “the behavioral aspects of the system and gauges usability, performance, efficiency, and security testing” (Hambling et al., 2019, p. 66). This was seen through being able to add, modify, and delete tasks, appointments, and contacts. White-box testing “looks at the internal structure of the system and investigates code and interface coverage” (Hambling et al., 2019, p. 66). This is seen as we look for coverage reports through the JUnit testing. Dynamic testing embodies all the other testing techniques as it exercises the program under test by engaging in the program. Testing related to changes involves the creation of a set of tests to demonstrate the system works as expected which is what was accomplished through the design process where length and null requirements were manipulated. The tests were also designated to throw an illegal argument exception if the criteria was missed. An example of code from these tests can be found below:

// Throws error if date is in past or description is too long

assertThrows(IllegalArgumentException.class,

() -> service.newAppointment(pastDate));

assertThrows(IllegalArgumentException.class,

() -> service.newAppointment(date, longDescription));

**Testing techniques not used**

The testing techniques that were not used in these classes were maintenance and static testing. Maintenance testing is keeping consistent on software as additional features are required or new faults requiring fixes are discovered. “Maintenance testing has three main goals: to find bugs in the released product, to diagnose any errors and fix them, and to ensure that changes made during post-release development are not causing problems with other parts of the system” (Softwatetestingo Editorial Board, 2021, para. 1). Since the software was built to accomplish the requirements then disbursed no intervention to fix or add on to functionality was necessary. Static testing involves looking over the code without exercising it. In this scenario it is impossible to do so because the point of the progress for the classes was to run them as Junit tests and make sure they adhered to the strict requirements given.

**Practical uses and implications**

These testing mechanisms could be used in many facets of software. They already are intertwined in the software development life cycle. Even on my other courses I have done a fair amount of dynamic testing to make sure the software can be run properly but also adheres to the standards given and the goals it needs to accomplish. White-box testing is prevalent throughout all software by utilizing statement coverage. “Statement coverage is a white box testing technique that ensures all executable statements in the code are run and tested at least once” (Imperva, n.d., para. 29). These techniques would best be used in the stages before complete release of a product so any issues can be ironed out before deployment. The exception to this list is maintenance testing which requires the program to be deployed and run then as software changes or user requirements advance, the software would need to have more features added in or corrected.

**Mindset**

While working through the project it quickly became obvious that the code needed to be interconnected to cover the functionality and test requirements necessary to receive a higher coverage percentage. It is important in the process of development to exercise caution as you must run continuous testing to ensure that the code functions. A key example of implementing caution that occurred was an endless loop created by insufficiently supplying an else statement on an if loop. With cautionary approaches and frequent bug testing, the code would have flowed much smoother before eventually finding the result through trial and error. As the Appointment class code is interconnected and was faced with an endless loop, I lacked proper functionality on the AppointmentService class due to this oversight to make it run efficiently. Being cautious when it came to this project involved compiling and debugging at every code block to ensure the functionality was as intended. An example of the loop that – while simple – done incorrectly wouldn’t set the proper variable for testing can be seen below:

// Sets criteria for update appointment date

public void updateDate(Date date) {

// Throw error if date is empty or in the past

if (date == null) {

throw new IllegalArgumentException("Appointment date cannot be empty.");

} else if (date.before(new Date())) {

throw new IllegalArgumentException(

"Cannot make appointment in the past.");

// Otherwise set date

} else {

this.appointmentDate = date;

}

}

// Return date

public Date getAppointmentDate()

{

return appointmentDate;

}

When approaching the requirements in the program though testing your own code you can become victim to bias. In a software developmental environment, this bias could prove detrimental in detailed bug testing. The developer may present that their code possesses no issues but could have also been victim to oversights. “Software developers and testers are more likely to choose positive tests rather than negative ones. This is due to the phenomenon called confirmation bias which is defined as the tendency to verify one’s own hypotheses rather than trying to refute them” (Calikli et al., 2010, para. 1). To combat this bias, I ran tests that purposely failed and would throw exceptions. This higher level of programming gave more of an insight into what could be wrong over only running tests that would pass. This would limit accounting for wild card variables. I saw this become an issue in a previous programming class where I had created a text-based dungeon crawler in java for a final project. It would ask for user input for their name at the start of the program, but because I did not account for input of a number into the string field, the name with numbers carried throughout the entire crawler. While not a hard issue to correct, this small oversight because I was only testing real names, could have been avoided.

When it comes to software development, it is like bringing a food dish to a potluck. In most cases you are contributing to the “meal” – the project that will incorporate your lines of code to put everything together. In another computer science course, I was introduced to Scrum teams. These teams divided into smaller groups to tackle the project into more digestible aspects. Each group took on user requirements and weekly met together to discuss obstacles and what went well. In the end the code from the groups was built together and tested before a final product was presented to stakeholders. Being disciplined in the commitment to quality and not cutting corners ensures that all the functionality is present as advertised. If the portion of code you have written cut corners and have faked out results to be considered done, it could negatively impact everyone that comes after you that are building from your code. Also, important to consider is the end user. Perhaps your program does not go through a development team to build on that may discover issues – you are the team – and what you deliver is the final product for an end user. It could prove disastrous if a company who has invested a lot of money into you and your skills puts out the product and it is riddled with issues from corner cutting. “In 2012 Knight Capital Group overlooked a very small software error that caused its software to process 4 million stock orders in an hour instead of spreading the orders over several days. The company lost $440 million, and its stock dropped 75 percent in a single day” (Farias, 2018, para. 13).

**References**

Calikli, Gul; Aslan, Berna and Bener, Ayse (2010). Confirmation Bias in Software Development and Testing: An Analysis of the Effects of Company Size, Experience and Reasoning Skills. In: *Workshop on Psychology of Programming Interest Group (PPIG)*, 19-21 Sep 2010, Leganes, Madrid, Spain.

Educba. (n.d.). *JUnit Code Coverage*. Retrieved from <https://www.educba.com/junit-code-coverage/>

Farias, H. (July 2, 2018). *Where you should never cut corners in software development.* Infoworld. Retrieved https://www.infoworld.com/article/3286265/where-you-should-never-cut-corners-in-software-development.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.

Imperva (n.d). *What is White Box Testing.* Retrieved from <https://www.imperva.com/learn/application-security/white-box-testing/#:~:text=White%20box%20tests%20can%20focus,known%20security%20threats%20and%20exploits>.

Segue Technologies. (August 7, 2015). *The Benefits of Adhering to Software Development Methodology Concepts.* Retrieved from https://www.seguetech.com/benefits-adhering-software-development-methodology-concepts/

Softwaretestinggo. (December 21, 2021). *Maintenance Testing in Software Testing.* Retrieved from <https://www.softwaretestingo.com/maintenance-testing/#:~:text=Maintenance%20testing%20has%20three%20main,other%20parts%20of%20the%20system>.