**Project Two Summary and Reflection**

Mike Martin

3/27/22

CS 320

SNHU

**Summary**

When discussing the approach I took to develop unit testing, it is important to have some understanding of what each file was required to do. The file I have chosen to break down is contact.java and contactService.java. Contact.java was required to allow up to 10 characters for the following variables: ID number, firstName variable, lastName variable, and phone number variable. Contact.java was also required to have an address variable, which allowed up to 30 characters. These variables must be hard capped at the required character input length and could not accept empty or null inputs. Next, contactService.java was required to allow the user to add, delete, and update contacts based on the unique ID. When updating contacts, the user should be allowed to change any of the variables listed above.

To stay aligned with the software requirements I wrote each test to emulate multiple inputs that the file was required to handle. For example, contact.java is supposed to reject an input if it contains null for any variable. One of my tests attempts to create a contact with each variable as null, switching the variable each attempt. I used this ideology when writing the unit tests for the remaining four files.

Moving on to the other four files outlined in the requirements documents: Task.java, taskService.java, appointment.java, and appointmentService.java. These four files are extremely similar when comparing their requirements to contact.java and contactService.java. Since the requirements are similar, this allowed me to reuse code from contact.java and contactService.java. For example; contact.java has a requirement of a 10 character limit for the firstName variable, both task.java and appointment.java are also required to have variables with specific character lengths. By reusing code I could ensure the remaining files followed the requirements given. I also reused code from the unit tests, to ensure that the unit tests worked and tested the respected files as intended.

The JUnit tests written for the above files are overall good quality. What I mean by this is that the JUnit tests have hit 80% test coverage of their respective files. The coverage indicates that the unit test file covers a percentage of their respective files code. For example; taskService.java is at 92.3% coverage, which means that taskServiceTest.java tests 92.3% of the files code successfully.

Throughout developing my code I ensured it was technically sound by following the best practices of Java. By this, I mean that I indented, used enters, used whitespaces, etc, according to Oracle’s “Code Conventions for the Java Programming Language”. To show this see lines 21 to 23 of appointment.java,

“**if(aDate == null) {**

**throw new IllegalArgumentException("Invalid Appointment Date");**

**}**"

I also ensured that my code was efficient by developing my code in a way that it could be reused, also known as modular. This allowed me to take code from other files such as task.java and create new files for other purposes; appointments.java. The following example shows a code snippet first from task.java, then from appointments.java.

“**public void setTDesc(String tDesc) {** *//Method to set the value of taskDescription and check if the passed value meets requirement*

**if(tDesc == null) {**

**throw new IllegalArgumentException("Invalid Task Description");**

**}else if(tDesc.length() > 50) {**

**throw new IllegalArgumentException("Invalid Task Description");**

**}else {**

**taskDescription = tDesc;**

**}**

**}**”

“**public void setADesc(String aDesc) {** *//Method to set the value of appointmentDescription and check if the passed value meets requirement*

**if(aDesc == null) {**

**throw new IllegalArgumentException("Invalid Appointment Description");**

**}else if(aDesc.length() > 50) {**

**throw new IllegalArgumentException("Invalid Appointment Description");**

**}else {**

**appointmentDescription = aDesc;**

**}**

**}**”

**Reflection**

While developing the JUnit tests I chose to stay relatively simple when designing the unit tests. Most of the tests reuse code from other tests, this helped keep it simple and easy to debug. I applied the simplicity mentality when choosing which testing techniques to use and I ultimately chose the following: assertTrue, assertEquals, and assertThrows. Using these three assert statements I was able to accomplish a total of 83.6% test coverage, which is high enough to ensure the files being tested do indeed follow the requirements document. On top of using those assert statement I also imported the related regular and service class files (eg. contact.java and contactService.java).

“**import contact.ContactService;**

**import contact.contact;**”

I also chose to create objects which would help test the setters and used a for statement to grab and assign the objects ID number.

“**ContactService contactServ = new ContactService(); //creates new object**”

“**for (contact cant : contactServ.listContacts().values()) {**

**testdelID = cant.getcontactID();**

**}**”

Since the technique above worked so well, I ultimately chose to stay away from many other techniques and assert statements. An example of some assert statements and techniques I chose to exclude are: assertFalse, assertNotNull, assertNull, timeouts, dynamic tests, and parameterized tests.

While the technique I used in this project is on the simpler side, it does not mean it can’t be scaled and used on larger projects. The choice on if to use my above strategy would come down to the complexity of the code, however that does not mean the simplicity mind set could not be applied. I believe that not over complicating your code is the best way to ensure your project works as intended.

As with anything else, it is always important to apply caution when needed. Within this project the only time I applied caution was when I noticed the code was getting complex. I used that caution to take a step back and look at the project with fresh eyes. Ultimately this allowed me to see how the unit test, the base class and the service class interconnected, which allowed me to simplify the code. As an example, I imported contact.java and contactService.java into contactServiceTest.java. This allowed me to create an object through contactService.java and then use a for loop to extract and assign the unique ID. In return the unique ID allowed me to test removing, adding, and updating contacts.

“**import contact.ContactService;**

**import contact.contact;**”

“**ContactService contactServ = new ContactService();** *//creates new object*

**contactServ.addContact("Mike", "Martin", "4446761212", "123 address");** *//add contact*”

“**for (contact cant : contactServ.listContacts().values()) {**

**testdelID = cant.getcontactID();**

**}**”

“**contactServ.deleteContact(testdelID);** *//deletes created task*”

The way I tried to limit bias was by relying on the JUnit test coverage. When you check the coverage of your unit tests it returns a percentage, 0% - 100%. I relied on that percentage over my own judgment, as I feel that is the best way to ensure that the code meets all requirements set in place by the client. Bias can be a problem, as it can get in the way of quality work. Bias tends to fool you into believing your code may be of high quality, when it may not run as indented. However, I feel that as long as you have proof, such as coverage percentage, then bias should not be an issue. The percentage will backup the claims of the developer or it’ll show the developer that the file is not ready.

It is important to never cut corners when developing software, cutting corners can create more issues in the long run and that’ll take more time to patch than if it was done the right way at first. Take the release of Android 12 on the Google Pixel 6 Pro as an example. The software on release had so many bugs that the phone was nearly unusable. It took about 4 months and a mountain of bad press before most of the release bugs were patched, and this excludes bugs created by the patches.

Overall I believe this project taught me a lot of practices and mindsets that I can apply to other parts of my life.

*Contents*. Code Conventions for the Java Programming Language: Contents. (1999, April 20). Retrieved April 17, 2022, from https://www.oracle.com/java/technologies/javase/codeconventions-contents.html

Hicks, M. L. (2021, December 14). *Pixel 6 bugs: Everything we've seen so far*. Android Central. Retrieved April 17, 2022, from https://www.androidcentral.com/pixel-6-bugs