### SNHU

### Project Two: Summary and Reflection Report

Andreas Galatis

### Neil Kalinowski

### CS 320 Software Automation & QA

June 20, 2022

**Summary and Reflection Report**

**Summary**

A static testing approach was undertaken to perform a unit test on the Classes in the Contact Service application, which consisted of three packages, ContactService, TaskService, and AppointmentService, and each had two classes to handle the various required functionalities that needed to be tested. For example, the Contact Class, in the ContactService package, needed to have a unique ID, first and last name, address, and phone number, with input limits on the length of characters allowed. To test for the functionality of these requirements, first a manual code review is performed to ensure that the corresponding functions to perform those requirements are written as specified, with exceptions to be thrown in case an incorrect input is entered, according to Java Object Oriented Principles (OOP).

Upon performing the manual code review for functionality, JUnit testing is performed to ensure that all possible scenarios are accounted for when the program is run. This is done by inserting specific JUnit commands that will account for various possible inputs. For example, since some of the requirements called for a certain input to be a certain length in characters, JUnit commands were used to determine whether the exceptions to catch those boundaries worked, by purposefully inserting incorrect character lengths into the inputs. This was done by creating additional Classes with JUnit coding specifically for testing all functionality. Below is an example of such a case to test for a new Contact object being created.

Graphical user interface, text, application

Description automatically generated

First we see that the contactClassConstructorTest() method tests to see that the application functions correctly when a new Contact object is created with all the correct information, such as an ID, first name, last name, phone number, and address. Once the new Contact object is created, the JUnit commands *assertAll* and *assertEquals* are used to make sure that when those objects are called with get methods that they truly match their respective entries. Furthermore, we see two more methods, the contactIdNullTest() and the contactIdTooLongTest(), which each test to see that an exception is thrown when an incorrect input is entered, such as a null entry or one that is longer than the allowed characters. This is done by using the JUnit command *assertThrows*, and then entering a null or incorrect value into the class constructor. The example above only shows the methods written to test for incorrect inputs for the ID field, but more similar methods for each required field were also created.

Once all the functional code is written and reviewed, and all the testing code is written as well, a JUnit coverage test is performed and examined to ensure that the test coverage is satisfactory. While it is not always possible to attain 100 percent coverage on JUnit tests, with this simple application, full coverage was attained. Below is a screenshot of the coverage report.

Graphical user interface, text, application

Description automatically generated**Table

Description automatically generated**

On the left we see an overall review of all the various methods in each of the classes, with a green checkmark indicating that they passed the JUnit tests satisfactorily. On the right we see a more detailed analysis of each class and what percentage of coverage was achieved, how many instructions were covered, and how many were missed. If we look at the Contact class, we see that all possible 133 instructions were accounted for in the JUnit test and none were missed as they all passed at 100 percent coverage. This coverage review is a useful to ensure that no instruction is missed in testing.

Furthermore, clean coding practices, with solid Java OOP were utilized to ensure that the code is technically sound, and minimal lines are used. This was done by creating two separate classes in each package, one to handle the constructor of a new object, and another to handle the functionality of that object. For example, in the ContactService package, the Contact class handles the constructor to create a new object with accessor and mutator methods. Below is an example of the Contact class constructor, as well as the mutator and accessor for the Contact ID field.

**Graphical user interface, text, application

Description automatically generated**

**Graphical user interface

Description automatically generated with low confidence**

Then in the ContactService class, which is part of the same ContactService package, a method is called to create a new Contact object, and new methods are also created for additional required functionalities of the application, such as updating one of the fields. Here is an example of the method to create a new Contact object by calling the Contact constructor from the Contact class.

**Text

Description automatically generated with medium confidence**

Following that, we wrote several other methods, such as the one below to update the required fields by calling on the accessor methods created in the Contact class. The following method is used to update a first name field.

Text

Description automatically generated

By utilizing these fundamental Java OOP practices and clean coding protocols, we were able to ensure that our application was technically sound and efficient.

**Reflection**

There are many software techniques that can be used in projects, and the most common ones are static and dynamic testing. Dynamic testing is done while the code is running, in a live environment, and usually incorporates test data. This type of testing is effective to discover vulnerabilities in security, as well as analyze how the application will perform in production. No dynamic testing was conducted in this project. As mentioned above, static testing was predominantly used for this project. Static testing consists of manual code review processes, as well as using certain frameworks such as JUnit to assist in the code review coverage. The testing done in this project relied heavily on the JUnit architecture which is composed of three major components called Jupiter, Platform, and Vintage. In this project the Jupiter package was used with assertions to check for required functionalities. Examples of some of the Assert methods used in this project were given above, as well as the coverage attained in using the Jupiter package.

In addition to the two common types of testing noted, one other type of testing that is common in software application is called use case testing. Common particularly in Agile development, use case testing is a way to describe a requirement of the application as it pertains to the user of that application in a story format or in laymen terms. Below is an example of a use case test.

Table

Description automatically generated

Sometimes after undergoing unit testing, the application may function properly, but it may not be what the user needs. That is where use case testing is useful, as it ensures that the application does what the user needs it to do. In the example given of the use case test, we can see that the tester is to log in to the site xyz.com and make sure that they see the promotional text in bold red color and that it should be in the middle top of the site. Such a test could not be verified with a unit test, only with a person going in and trying it out to see that the result is what is expected.

In acting as a software tester, a cautious mindset is important, as well as anticipating all possible circumstances, as much as one can. The tester is the last measure between an application reaching production, before finding out that some important element was neglected. Testing should not be conducted on the end user, as this may cause the user to lose confidence in the application and eventually cost the company a lot of money to remediate the issue. For example, in June of 2021, Porsche had to recall almost all if its Taycan models, their first all-electric vehicle. Over 43 thousand cars had to be recalled all over the world, due to a sporadic error that triggers an emergency algorithm, causing the power to suddenly shut down (Russian Federation, 2021). While the error was a hard one to spot, it is the tester’s responsibility to make sure that all possibilities are examined.

As a tester and software developer, it is important to remain disciplined in the software engineering field, as well as removing all bias when coding. Since there is often no real accountability for the work a coder does, it is important for the one writing the code to be highly responsible and self-accountable. This can be achieved by engaging in peer code reviews, as well as have a continuous plan to keep improving one’s coding practices. It’s also important to go back and reflect on past work and find areas of improvement for future projects. This is the only way the field of software engineering will improve and become a true engineering endeavor. This calls for a high degree of integrity and discipline from all coders and testers, as well as commitment to quality as a software engineer professional.

**References**

Russian Federation : Porsche recalls almost all Taycan electric cars released A programming error forced Porsche to recall 43 thousand Taycan around the world. (2021, Jul 06). *MENA Report* <https://ezproxy.snhu.edu/login?qurl=https%3A%2F%2Fwww.proquest.com%2Fwire-feeds%2Frussian-federation-porsche-recalls-almost-all%2Fdocview%2F2548735611%2Fse-2%3Faccountid%3D3783>