**Project Two – The Importance of Testing Software**

Guilherme Pereira

Southern New Hampshire University

CS-320: Software Test Automation

Angelo Luo

August 13, 2021

**Summary**

Grand Strand Systems is a hypothetical software engineering company that has tasked me to create a mobile application that a customer could perform contact, task, and appointment services. For each of these services, a requirement was followed by what the service should do as well as what it is limited to do. For instance, the appointment service required that the address field have no more than 30 characters in length.

To test this software requirement, a field check is passed for every field entered and returns a Boolean value: true (pass) or false (fail) depending on the requirement of the field. J-Unit tests were created to test these different functions by using the assertTrue() or assertFalse()methods to confirm if the returned value was true or not. While testing for the success of each requirement was important, verifying that the output failed correctly if something did not meet the requirement was also just as important. If an address is successfully able to accept a string of less than 30 characters, but also more than 30 characters, then something is wrong with the code. For each J-Unit pass test there was also a J-Unit fail test, in which the assertFalse() methods were used to ensure that the function failed correctly. Additionally, a function’s intended requirement was also tested to see that they completed their task. The Task class has a requirement that tests whether a function can successfully delete a Task object from the list. Below is an example of the code written to test for success and failure:

### Task Delete Function (Pass)

@Test

**void** testTaskServiceDeletePass() {

String taskDescription = "Botanist will harvest flora on grassy plots";

Task taskDemo = **new** Task("003", "Harvest", taskDescription);

*assertTrue*(taskService.addTask(taskDemo)); // Validate add

*assertTrue*(taskService.deleteTask("003")); // Validate delete

*assertTrue*(taskService.getTask(taskDemo.getId()) == **null**); // Validate that the removed object is null.

} // Successfully passed

### Task Delete Function (Fail)

@Test

**void** testTaskServiceDeleteFail() {

String taskDescription = "Botanist will harvest flora on grassy plots";

Task taskDemo = **new** Task("006", "Harvest", taskDescription);

*assertTrue*(taskService.addTask(taskDemo)); // Validate if the objects was added

*assertFalse*(taskService.deleteTask("007")); // Should fail as no object with this ID exists

} // Successfully failed

My experience with J-Unit testing was great but felt limited at times. While I finally learned how to test my code, aside from peer-review and debugging the code endlessly, I still feel like there was more to J-Unit testing than just asserting whether a returned value was true or false. In testing, it makes sense that if something succeeds then you want it to return true, while if something fails you want it to return false. I feel like the code could have been further tested with the automated tests we learned about earlier this semester. These tests may have been able to cover areas I did not think about, such as testing for data types, testing for no input, and more. However, when it comes to the validity of my tests for this project, I do believe that my code was technically sound. As each test has a pass and fail associated with it, I correctly tested whether the code met the requirement or not. Once everything was completed, all the requirements were successful.

As for the efficiency of my code, it is difficult to say how efficient it was. As a developer and student, I am constantly striving to make corrections to my code to improve readability and efficiency. For this first Contact assessment, my code went through one additional software check to meet the requirement compared to the provided solution. Learning from my mistake, I took the previous solution into account to check for my validity and improved the code by having one less check. For the project one submission, I went back to my original Contact class code and decided to fix the readability and efficiency. Below is a demonstration of how the readability and efficiency improved. The first code body is taken from the original Contact constructor while the second is taken from the project one submission for the same constructor:

### Original Contact Constructor

**public** Contact(String id, String firstName, String lastName, String phone, String address) {

**if**(id.length() > 10 || id == **null**) {

**throw** **new** IllegalArgumentException("< Invalid ID >");

}

**if**(firstName.length() > 10 || firstName == **null**) {

**throw** **new** IllegalArgumentException("< Invalid First Name >");

}

**if**(lastName.length() > 10 || lastName == **null**) {

**throw** **new** IllegalArgumentException("< Invalid Last Name >");

}

**if**(phone.length() != 10 || firstName == **null**) {

**throw** **new** IllegalArgumentException("< Invalid Phone Number >");

}

**if**(address.length() > 30 || address == **null**) {

**throw** **new** IllegalArgumentException("< Invalid Address >");

}

// If none of the statements ran true, a new contact is created.

**this**.id = id;

**this**.firstName = firstName;

**this**.lastName = lastName;

**this**.phone = phone;

**this**.address = address;

### Revised Contact Constructor

**private** Contact() {}

**public** Contact(String id, String firstName, String lastName, String phone, String address) {

**boolean** validId = validate(id, 10);

**boolean** validFirstName = validate(firstName, 10);

**boolean** validLastName = validate(lastName, 10);

**boolean** validPhone = validatePhone(phone, 10);

**boolean** validAddress = validate(address, 30);

**if** (validId && validFirstName && validLastName && validPhone && validAddress) {

**this**.id = id;

**this**.firstName = firstName;

**this**.lastName = lastName;

**this**.phone = phone;

**this**.address = address;

**return**;

} **else** {

**throw** **new** IllegalArgumentException("< Invalid input >");

}

}

**Reflection**

The testing techniques I used in this project are debugging, creating J-Unit tests, using assertTrue() and assertFalse() methods, and using a lambda expression for the assert.throws() method. Debugging is the process of running the code and checking to see if any errors occur from what is written. Sometimes I would create an object or throw in a print statement to ensure everything is printing out correctly or printing at all. J-Unit tests are a feature in Eclipse that specifically test for a code in a specific class using an @Test annotation, which will show if the test passed or failed on the testing window. J-Unit tests display how many tests were performed and if something passed, failed, or a mix of both. The assertTrue() and assertFalse() methods would return true or false depending if the Boolean retrieved matched the respective function. These were important to test for code that passed successfully or failed correctly – assertTrue() and assertFalse() respectively. Finally, the lambda expression using the assert.throws() method was used to help create objects that would otherwise fail and be unable to be tested for during compile time.

Bias was limited in this project by ensuring that all the software requirements were followed to the word. While success is what the Grand Strand System is looking for during these tests, I believe it was also important to test that the code could also fail if it did not meet the requirements listed. Therefore, by checking for both results, I limited my own bias that the code is good and works. What I could have done to further reduce bias was to have someone else create their own tests or create their own objects and see whether or not my project worked as intended or if the software requirements were correctly met.

Discipline is important in almost every aspect of programming, but it weighs heavily to be disciplined in testing and quality assurance because of the impact poor code could have on others. A great example is from an article one of my classmates shared with us during this week’s discussion regarding the failure of a poorly tested machine which resulted in the death of 6 patients. The Therac-25 provided radiation treatment to cancer patients, but some were more unfortunate than others. For instance, the machine ran into a byte overflow error that would switch its treatment mode from a light radiation beam to a more intensive beam, which would deliver a lethal dose of radiation. The following statement explained by Lynch shows just how the machine ran into this error:

During verification, the Class3 variable determines whether the hardware is configured correctly. A non-zero value indicates failure, whereas a zero value indicates that everything is setup correctly, and that treatment can proceed. Because the setup code ran hundreds of times and a byte can only hold 255 possible values, on the 256th attempt of setup, the shared variable would be set to 0.” (Lynch, 2017).

Additionally, the Therac-25 was created using code previously used for the Therac-20, which resulted in problems with the new hardware the software was working on. This is one example of how cutting corners affected the program’s performance and failed to provide additional safety measures for the patients under this machine. My role as a software developer would be to ensure my code is efficient, well-tested, and technically sound. While I will begin to write more tests in the future to check the validity of my code, I would also like to call upon colleagues or other co-workers to ensure the program works as intended. In an agile setting, I would like to communicate the program with the Product Owner and possibly the clients to ensure that the requirements being asked of are correctly written and performed.

**References**

Lynch, J. (2017, September 19). *The worst computer bugs in history: Race conditions in therac-25: Bugsnag blog*. The Worst Computer Bugs in History: Race conditions in Therac-25. https://www.bugsnag.com/blog/bug-day-race-condition-therac-25.