CS320 Project Two: Summary and Reflections Report

Gary Clark

Southern New Hampshire University

CS320 Project Two: Summary and Reflections Report

At Grand Strand Systems, we strive to deliver secure, efficient, and logical back-end services to our clients. While developing these solutions, it is necessary to thoroughly test our product before delivery to ensure it meets both ours and our client’s standards. Recently, we have delivered a mobile application solution to a client which enables contact, task, and appointment management services and to ensure the products reliability, we employed a technique called JUnit testing. In this method of testing, each application is broken down into the smallest possible functional units to test many individual behaviors. If these features behave as intended, then we can ensure the program will behave as intended, giving confidence to our clients that we provide some of the most exceptional back-end services on the market.

The process of implementing JUnit testing starts with first identifying the projects requirements which will be used to build the structure and plan for the testing phase. In the contact application, we were given various types of data to track as well as the formatting requirements for each attribute. As such, we had to ensure the JUnit tests verified that our application delivered upon the client’s requests. Each attribute within the application was given a maximum length requirement in addition to the avoidance of null fields, and the application was required to modify certain attributes to create, update, or delete a contact. To verify the intended functionality, we included two test classes to observe the behavior of both the structure of the contact and the contact service. Within the contact class itself, there are certain tests built in to check information which is received as input, and the test class would monitor the application for “throws” or instances where it successfully caught incorrect entries. It is not enough to just put in a single contact and validate the data initialized correctly, instead we must test both valid and invalid entries for each attribute. This allows us to observe that the application can handle a wide variety of inputs and maintain the expected behaviors. For instances where correct information was received as input, the test class used a method called “assertTrue” in which the function compares the value of the attribute to the expected value of the attribute and asserts the statement as true if they are equal. Once we can confirm that the contacts can be initialized properly, the focus must turn to the required functionalities regarding the modification and addition of these contacts. Much in the same way the JUnit tests verify correct inputs to the contact class, the test class for the contact service uses the same “assertTrue” method to validate that the contact was initialized and added to the map which tracks all contacts in the system. However, we must validate other scenarios of which the application will encounter once deployed and we can surely assume that there will be more than one contact within the deployed system. Therefore, we must repeat the same test for the addition of multiple contacts to the application and verify that each contact was initialized properly. Also, consider a contact list with many contacts in storage, what would happen if someone tried to enter information which was already in the list? Would the system duplicate the contact, replace the contact, or just crash completely? Our solution was to include a check before the addition of a contact to the system to search for an existing contact with that ID and this allowed us to design a JUnit test which would listen for thrown exceptions when there is a match. When switching focus to updating an existing contact, we must first initialize a contact and confirm that the data is correct. This contact will then be modified using the methods built into the service class, after which the test class can validate the updated data fields against the expected values. Finally, when considering the ability to delete a contact, our application must first initialize a contact to delete. Once that has been done, the application can call the “removeContact” method and this method finds an existing contact with a provided contact ID. Should the application find the contact, it will use the remove functionality included in the Java Maps Utility library and our test case will validate that the resulting entry is now null.

The testing methodology for the Task and Appointment applications was nearly identical with only minor modifications made to account for differing functional requirements. Within the Task application, the only major differences came with the formatting requirements of the attributes which were accounted for by simply modifying the validation check method with the correct requirements. As such, the testing structure was nearly the same with tests validating task initialization and service application functionality. Our only major adaptation came within the Appointment application which added the need to implement a date attribute to the object which could not be in the past. Therefore, whenever the application is initialized, we must first get and store the data and time for the validation check to compare received inputs against. These functionalities required that we add multiple JUnit tests to verify behavior for both a date in the past and a date in the future. Once again, the “assertThrow” method fits the bill with invalid dates throwing an exception which the testing class can see. In our service class, we also implemented tests to verify that these inputs behaved as expected even within lists of multiple appointments. The success of a JUnit test ultimately relies on the end functionality of a product, but a key metric which allows developers to evaluate how thoroughly they are testing requirements is known as the coverage percentage. This percentage evaluates what values and operations were successfully tested and which were missed by the tests. Within this application we were able to achieve an aggregate coverage of 40.3% with that increasing to 58.3% excluding the test classes. The latter is more accurate of the final behavior of the application as the test classes will not be a part of the deliverable to the client. Our testing should be considered a success as the structure of the program is simple and the expected input values are limited. Ideally, we would strive for at least 70% coverage across the board, but given the nature of the project, we can verify the expected functionality with our results. The contact class was somewhat of an outlier as you can see in the images section at the end of the paper as the coverage of the contact package was only 48.2% (Image 4). This is far lower than the 62.8% and 64.3% of the appointment and task packages respectively. The design of the code combined with the JUnit tests gives us great confidence in the product we are delivering to the client.

One aspect of designing successful JUnit tests that is often overlooked is that successfully testing an application requires that the application be efficient and technically sound. Developers who utilize best coding practices and naming conventions will have a much easier time designing tests with higher levels of coverage than those whose code is messy and confusing. In our applications, we emphasized the structure of the code within the application to align with best practices for secure and efficient code. An example of the secure coding methods used lies with the implementation of the length requirements for each attribute. We created final integers variable types with a private level of accessibility to make it more difficult to pass false or modified information through the system (Image #1). This secure minded approach to systems design continues into the constructor methods of each application where each input is validated before assigning the value to the object (Image #2). By including exception handling within the constructor, we can prevent unintended system errors which minimize the chance of invalid data entering the system. With all the focus on security, it was also important not to sacrifice efficiency in both the application function and the test cases. The chosen data structure to store objects was a Hash Map which is a more efficient and lightweight structure to search and add items to (Image #3). The key for each item was it’s ID which also contributes the security of each application. As we implemented the JUnit tests, efficiency was considered, and certain tests were combined into one to save time and resources. The most prominent example of this was with the tests involving multiple items added to the list and we realized it was possible to test that functionality as well as the remove an item function. This saves time and space while keeping the code clean and organized, which are key aspects of a successful JUnit test application.

This foray into the world of JUnit testing has been quite beneficial to our overall understanding of quality assurance and testing as it applies to our products. While static and dynamic testing methods have been used consistently throughout the projects within our organization, there are more specialized testing techniques such as JUnit testing that can better assist with certain projects. These methods can often be used in conjunction with one another to further enhance their effectiveness. Within the test design process, we formulated some of our JUnit tests from the usage of the Error Guessing technique. In this method, we aimed to guess which errors would be likely to occur within this application and we used this information to design JUnit tests to expose these errors. The most obvious of which was the likelihood of accidental inputs which did not align with what the application was designed to take. Therefore, we formulated tests around the idea that these inputs will happen and in the JUnit test we ensured these extraneous inputs were handled correctly. Besides these methods, there are still many others which given the right circumstance would prove very beneficial to implement. Some of these include State Transition Testing, Decision Table Based Testing (DTBT), and Boundary Value Analysis (BVD). During state transition testing, testers will enter various input conditions in sequence to observe the behavior of the application such as entering a password incorrectly three times. In DTBT, we would break down the application into a set of rules of which there is an input and one or more outputs. Finally, in BVD we would place our testing focus on the boundary values of a range of inputs. This would allow us to observe the behavior of the system when information is on the boundary of valid and invalid which is the most likely spot for certain errors to occur. Any of these methods require an air of caution as it can be easy to overanalyze certain situations and this cautious approach was carried over into our most recent project. While considering which inputs and scenarios to design JUnit tests for, we had to be careful not to overcomplicate the solution. For example, is it possible that someone would enter the enter Encyclopedia Britanica into the description field of the task? Sure, but is it likely? No. There are a near infinite number of scenarios of which could potentially occur, but it is impossible to test each one. Thus, it is our job as testers to analyze which events and errors are most likely to occur and design an effective method of testing for those errors. We must consider the interactions between the methods and code within our system, and we must also try to remain as focused on the bounds of our application as possible. Should the scope of testing become too large, many smaller, more likely issues will be overlooked. This cautious approach also applies to our inherent biases that arise from reviewing our own product. An early version of these applications included a storage method of an ArrayList for the items which is far slower to search through than a Hash Map. When we discovered this method, it was hard at first to change the code as it was our own solution, and it did work. However, as developers we cannot let our biases prevent us from delivering the best product possible and in this case refactoring the code to use a map data structure was the more effective solution.

It is our duty as software engineers to commit to quality as we design our products. Cutting corners will only lead to dissatisfaction and through the nature of developing a service for others; we would be negatively impacting the lives of others. Besides the moral obligation to always strive for good, there exists the legal and financial obligations to the client and the organization. Half-finished products will create great liabilities in both of those areas which may lead to nothing or may lead to multibillion dollar lawsuits in the case of multiple Silicon Valley giants. Therefore, it is our duty to remain focused and diligent while practicing thoughtful and thorough software for our clients and organizations.

Reference Images

#1

Text

Description automatically generated

#2

Text

Description automatically generated

#3

Graphical user interface, text

Description automatically generated

#4

Table

Description automatically generated with medium confidence

#5

Table

Description automatically generated

#6

Table

Description automatically generated with medium confidence