### Summary and Reflections Report

1. **Summary**
   1. Describe your unit testing approach for each of the three features.
      1. To what extent was your approach **aligned to the software requirements**? Support your claims with specific evidence.

A practical way to ensure that user needs are being correctly met in a software project is to test against those exact requirements. This is precisely the approach I took when developing the mobile application as a member of the development team at Grand Strand Systems. Our client had requested a system that would allow users to manage their contacts, tasks, and appointments. Every week I was provided a list of requested features for each part of the system, starting with the contact subsystem. According to the specifications, the contact object itself was required to have five fields – a unique id, a first name, a last name, a phone number, and an address. The id, first name, and last name fields were to be no longer than 10 characters long. The address field was to be no longer than 30 characters long. The phone number, however, was to be exactly 10 digits. None of these fields were permitted to be null, and all of them were required to be updatable after creation except for the unique id.

In order to align my approach with these software requirements, I wanted to introduce unit tests that would verify that the contact fields would adhere to each provided specification. First, I created the constructor for the contact class that required values for all five of the aforementioned fields to be supplied and then assigned them to the object’s member variables. At this point, anything was allowed to be assigned and the requirements were not yet being met.

My next step was to write unit tests for the contact class’s id field. Specifically, I identified that I would want to test contact creation with a null id, with a zero-character id, with a nine-character id, and an 11-character id. When writing these tests, I coded for what I wanted to happen, despite knowing the functionality was not yet present in the contact class’s constructor. In other words, I expected that creating a contact with a null id would throw an error in my test, even though I had not yet coded the constructor to do such a thing. My tests expected that zero, nine and 10-character ids would not throw an error, but an 11-character string would. When I ran these tests for the first time, only three out of five of them passed. The two test cases that failed were the null id and 11-character id cases, because as I said before, the contact constructor was not yet coded to refuse those types of values.

My next step was to implement error checking in that constructor. I added checks against the id argument provided in the constructor and returned an illegal argument exception with a relevant message back to the caller if the id was null or if the id had a length greater than 10 characters. When I ran my tests again, I now saw that all of them passed. This was my process for ensuring that my approach developing this application was aligned to the provided software requirements. For every given requirement I would write unit tests with appropriate test cases before even implementing the necessary behavior in the source code itself. It was only after that step that I went back to start developing the feature in the source code. Afterwards, running the tests again would let me know if the requirement had been met. I stuck to this process throughout the lifetime of the project.

* + 1. Defend the overall quality of your JUnit tests. In other words, how do you know your JUnit tests were **effective** based on the coverage percentage?

I believe my unit tests were high quality because they were small enough to focus on only one requirement at a time, and because they all reported the expected results. Moreover, I consistently ran coverage reports to see which lines of code in my project were being tested. The value in doing this cannot be overstated. My coverage reports occasionally showed me that there were a significant amount of execution branches in my program that were simply never tested against. My overall coverage percentage was a helpful metric to quickly sum up how many lines of code were executed between all of my unit tests. This metric can be great for generalizing how much of a project has been tested. For larger projects, it is not necessarily always feasible to achieve 100% testing coverage for a variety of reasons, most of which boil down to prioritization of resources. Given that this mobile application was relatively small, however, I felt that it would be excellent practice working to achieve 100% testing coverage. I was able to reach that metric, ensuring that every line in my project was tested in some capacity. This does not prove that my program is 100% error-free, however!

* 1. Describe your experience writing the JUnit tests.
     1. How did you ensure that your code was **technically sound**? Cite specific lines of code from your tests to illustrate.

I wanted to ensure that my code was technically sound, so I made a conscious effort to follow standard best practices for software development. The easiest way to do so was to use comments very liberally throughout the source code. I documented nearly every function in my program, explaining what each parameter was used for during the body of a function, as well as any requirements for the arguments being supplied while calling the function. Specifically, the aforementioned contact class had a thorough comment before the constructor explaining the length requirements of every field to be supplied. I also used comments to indicate which functions may potentially throw errors if certain conditions were not met. On top of that, I kept redundant code to a minimum by extracting common or repeated logic into their own reusable functions. The contact class fields id, first name, last name, and address were all very similar in that none of those values were allowed to be null, and that they all had a limit on how many characters could be present. Rather than checking these individually, I introduced a class functional with the signature validateInput(string fieldName, string fieldValue, int limit). This allowed me to simply just pass in the name of field being checked, the value being supplied, and the maximum character limit for the field. I was able to use this function multiple times to quickly ensure that each field passed into the contact constructor met the requirements, and I was able to take advantage of the fieldName parameter so that I could throw errors with more detailed messages! Other ways I followed industry standard best practices was sticking to camel case naming conventions throughout the project and making sure that every variable declared was actually used.

* + 1. How did you ensure that your code was **efficient**? Cite specific lines of code from your tests to illustrate.

In addition to keeping my code technically sound, I also wanted it to be efficient! Initial drafts of my program included some less than ideal approaches to handling certain requirements. For example, my first pass at verifying a phone number for a contact was to iterate through every single character in the string and check whether or not it was a digit. If a non-digit character was encountered, an error would be thrown back to the caller. If no non-digit characters were found, then the string was assumed to be valid. Although this approach did work, it could have been better. I replaced this code with the matches() function from the Java library. I was able to instead supply a regex string specifying the regular expression that the phone number should match. My entire loop and individual character processing statements were replaced with just one line: if (!phone.matches("\\d{10}")). This was a much more efficient way to check if the phone number string was a 10-character string of digits! Another area for improvement was my initial handling of the contact service and task service classes. Originally, when a new contact or task was created, their related service class would store them in an ArrayList data structure. This meant that in order to add a new entry to the list, every element in the ArrayList would first have to be checked for a matching id. Likewise, deleting an element would require traversing every element until it was found (or even worse, not found)! This created a situation where the execution time to add or delete a contact or task to its associated list was directly related to how many elements were already in that list, potentially requiring many checks. To make this more efficient, I replaced the ArrayList with a HashMap, which uses a unique key to identify containing elements. Instead of iterating, I was able to simply attempt to insert an element with a specific key into a HashMap. If it was successful, then it was the only element with that key present. If it was not successful, then an element already existed containing that key! Deleting an element from the HashMap also benefitted from this same simplicity. In these ways, I ensured that my code was efficient.

1. **Reflection**
   1. Testing Techniques
      1. What were the **software testing techniques** that you employed in this project? Describe their characteristics using specific details.

Unit testing was the main focus of this project. For every module, I planned out my tests based on the given requirements. I would write a test, run it to ensure it failed, and then go back and introduce the functionality into my code to make the test pass. I continued this process for every requirement, refactoring my code as needed and running my tests again to ensure they were still passing.

Within my unit tests, I used boundary analysis by testing multiple values around each requirement to verify that they worked as expected. For example, the id field of each class was to be non-null and a string less than ten characters. In order to use boundary analysis here, I created the boundaries of strings with nine, ten, and 11 characters, then tested to make sure nine and ten length strings would correctly be assigned but strings of 11 would return an error to the caller.

Finally, I used informal review as a technique to improve the quality of my project. After each submission, I would receive feedback from the professor indicating potential areas of improvement. As code quality and efficiency is important to me, I made sure to constantly update my project include these recommendations!

* + 1. What are the **other software testing techniques** that you did not use for this project? Describe their characteristics using specific details. **<Write your answer>**

I also did not have to use any strictly experience-based techniques, such as error guessing or exploratory testing. The requirements of the milestones were always explicitly stated in an easy-to-understand format, and the use cases of each were easily inferred by the accompanying prompts. The experience-based techniques above can be useful for catching certain errors that might not inherently be obvious by performing unit tests or reviewing the code itself. For example, a developer might know from experience that there is a bug in a popular library or framework that can lead to unexpected behavior given certain conditions. They might know how to work around that to improve the reliability of the application.

Additional techniques that were not used for this project include integration testing, system testing, and stress testing. These types of testing are more applicable for larger applications or applications that expect to handle a large concurrent user base. Integration testing can be used to determine how the various components of an overall system behave when they interact with each other, which can demonstrate how well the individual pieces work together. System testing is a rather complex task of evaluating the entirety of the system, which can include its components, use cases, data storage, runtime environment, and even more. The implications of system testing are massive, but to summarize, it can be used to catch any errors that might have been missed while evaluating the pieces of a system in isolation. Stress testing can be used to see how the system will behave under heavy loads. If a company anticipates that their service might be used by 1000 users simultaneously, they could simulate that amount of activity on their system and measure how well their system holds up. They might be interested in finding out how quickly the system can respond to individual users under that amount of load, or they might even test to see if they could potentially be made vulnerable to denial-of-service attacks.

* + 1. For each of the techniques you discussed, explain the **practical uses and implications** for different software development projects and situations.

I made use of both black box and white box testing techniques during this project. Black-box testing is said to encompass the what of a system as opposed to the how. These types of techniques work on a specification or a model of a system. Functional testing, which is the process of making sure that the project is functioning as per the requirements, is another way to utilize black box testing techniques. The overall practical use of black box testing is to verify what outputs are produced given certain inputs without knowing the exact implementation details. This can lead to a more comprehensive series of test cases, as if the tester is unable to make any assumptions based on the code itself, they are less likely to supply biased input values.

Several practical ways to make use of black box testing are by performing equivalence partitioning and boundary value analysis, both of which I did for this project. Boundary value analysis is the process of testing multiple input values around the “boundaries” of a unit. This can help catch any potential miscalculations or edge cases. Equivalence portioning is the process of dividing a collection of inputs into various domains and testing a few values from each to extrapolate how the unit behaves with values from each domain. For example, a function that accepts a whole integer might test the domains of negative integers, the number zero, and positive integers.

White box testing focuses instead on the internal structure of components within a system. It does make use of the actual code used to develop the application in order to determine the application’s adherence to the specification, or to calculate how much of the code is evaluated during unit testing.

Test coverage reporting and complexity analysis are two of the white box techniques I used. When writing unit tests for a software product, a test coverage report can be generated which will calculate what percentage of the lines or branches in a program have been executed by the collection of unit tests. While this does not prove a lack of bugs in a program, it can show that code has been tested to some degree, which can increase confidence or reduce bugs. Complexity analysis is the process of reviewing the source code for areas that may prove to be suboptimal, inefficient, or error prone. For example, a developer might notice that there is a more efficient algorithm they could use in place of the existing code, and then refactor that portion of the code. They could then run the unit tests again to ensure that their code is still functioning as expected with the changes!

* 1. Mindset
     1. Assess the mindset that you adopted working on this project. In acting as a software tester, to what extent did you employ **caution**? Why was it important to appreciate the complexity and interrelationships of the code you were testing? Provide specific examples to illustrate your claims.

The scope of the mobile application was a bit small, and the contact, task, and appointment modules were not necessarily interconnected to a significant degree. As a result, I did not feel the need to employ caution in the same way I might with larger projects, but there are still a few points I can speak to.

For example, the constructor for the contact class needed five arguments to be supplied: the id, the first name, the last name, and the phone number, and the address. Considering that each of these parameters had requirements for their allowed values, it meant that just calling the constructor for a contact object would lead to five potential points of failure. I felt obligated to do a lot of unit testing for this to ensure that the program would provide meaningful output to the caller should it fail.

Another important consideration was input validation. The contact, task, and appointment classes often had member variables that needed to be not null and also needed to be a string under a certain length. I eventually introduced a function called validateInput(*String* *fieldName*, *String* *fieldValue*, *int* *limit*) that would return true if the input was valid, or throw an error if it was not. Ideally, I would have extracted this function into a file such as lib.java and imported it into Contact.java, Task.java, and Appointment.java so that I could reuse it. For the purposes of this assignment, however, I was not supposed to introduce any additional class files, so I simply copied and pasted the function into each class individually. In doing so, I would have been able to take advantage of code reuse. In larger projects, it is important to identify common functionality that can be extracted outside of individual classes and instead imported in to all the classes that need it. If the code is simply copied and pasted, and a bug or error is identified later, it means that all instances of that copied function need to be found and corrected!

* + 1. Assess the ways you tried to limit **bias** in your review of the code. On the software developer side, can you imagine that bias would be a concern if you were responsible for testing your own code? Provide specific examples to illustrate your claims.

One way I tried to limit bias was by writing a lot of tests to check my implementations. I did not want to just write one passing test and consider the feature complete. Instead, I used techniques such as boundary testing by testing multiple values at and around the decision branches where the result of the function should change.

Another way I tried to limit bias was by testing unexpected values. For example, I would frequently pass in null values or empty strings to see how my code would behave.

I believe that bias is a huge concern when testing code self-written code. It can be far too easy to assume that your code will only be used in a certain way, but it is entirely possible someone on the team may see the code and attempt to use it in unintended ways. If the code is not adequately documented or if it does not return appropriate error messages, that developer may not even realize. Tests are great to verify that features work as expected, but tests should also verify that code should fail as expected if use conditions are not met.

* + 1. Finally, evaluate the importance of being **disciplined** in your commitment to quality as a software engineering professional. Why is it important not to cut corners when it comes to writing or testing code? How do you plan to avoid technical debt as a practitioner in the field? Provide specific examples to illustrate your claims.

It can be difficult to constantly hold oneself to a standard of quality, but I believe it is more difficult to deal with the aftermath of a messy code base. When I first started writing code, I was so excited to make things happen that I did not consider the implications of my design choices. More often than not, this would lead to situations where I was able to add features quickly early on, but I would eventually reach a point where adding more was virtually impossible. The code would not be modular enough to support reuse. The design would be so rigid that it could not be adapted to support additional use cases. Modifying code in one place would potentially break code in other places with no obvious connection. I needed to start approaching development in a different way.

Today, I am aware that software development is not only about writing raw code. It is just a small part of a larger process that also involves requirements gathering, system analysis and design, planning, documentation, and testing. I have learned that sometimes writing new code is not the answer, but rather the solution is to take a step back and evaluate how the new feature fits within the existing project structure. Is this feature related to any others? Can this feature be abstracted in such a way to take advantage of already existing code? What are the future implications of this feature? Can this feature be abstracted in such a way that upcoming features can take advantage of the new code that has to be introduced? Cutting corners is a trap. Best case scenario, the code introduced works perfectly in its intended context. Worst case scenario, project overall becomes less flexible, harder to test, harder to debug, and more time will be spent down the line to try to fill in these corners later. It is not worth it.

There are a few ways I try to approach my projects as to limit technical debt:

1. Clearly define the features I am working on. What functionality needs to be present for the feature to be complete? What functionality does not need to be present at this current time?
2. Identify any pre-existing features that may share similarities between the new features. What can I leverage to introduce new features? Can I refactor old code to support new use cases?
3. Before developing the feature, write the associated tests for it. This will help create a clear point where the definition of done be can considered met.
4. Implement the features and ensure all of the unit tests pass.
5. Once the features have been implemented and the tests pass, refactor the code! Review the code that was just written and look for any instances of code repetition, unclear variable naming, or inefficient algorithms/operations.

References

Hambling, Brian, et al. *Software Testing : An ISTQB-BCS Certified Tester Foundation guide - 4th edition*, edited by Brian Hambling, BCS Learning & Development Limited, 2019. *ProQuest Ebook Central*, http://ebookcentral.proquest.com/lib/snhu-ebooks/detail.action?docID=5837074.  
Created from snhu-ebooks on 2023-08-29 17:06:55.