Justin Phillips

Mike Alesso

CS-320 Software Testing Automation

Southern New Hampshire University

8/19/2024

Summary and Reflections Report

1. Programming and Static Testing Development Summary

This mobile application required analysis of testing methods and strategies in order to find well-suited tests for the sake of efficiency. Our tests were structured around the software requirements and executed using static testing JUnit scripts to ensure the quality and performance of every function and routine in its output; Visual code inspections were also employed to ensure the quality of each function in its design. The software requirements were the production of three distinctive modules, each module composed of an Object class and a Service Object manager class. These modules, the Appointment, Task, and Contact modules, were of a similar composition with a list of necessary attributes and acceptable qualitative parameters, i.e. field length. For each Appointment, Task, and Contact Object, we met ten character unique ID parameters which acted as the keywords for each object in its Service neighborhood. Their major differences were in their exact parameters and functions.

The Appointment’s properties were a 10 character ID, a Date date either today or in the future, a String description not to exceed 50 characters, and no null fields. The ID was also not to be updatable after the creation of the object. Java Eclipse, the IDE of choice, was unhappy with calls to an outmoded java.util.Date reference, so this deprecation warning was suppressed utilizing the relevant in-editor annotation.

****

I believe this is a good example of intentionally not testing for something our IDE has static testing for by default. The outmoded java.util.Date library was functionally relevant to us regardless, and beneficial to employ for this object due to the intuitive nature of the Date layout and construction. To create a safe constructor, I recalled the functional setter methods during the construction of any new Appointment.

A screenshot of a computer code

Description automatically generated

This prevents redundant code and allows the checker to always run, whether on initialization or a future update, in an identical manner each time. This technique recurred over each parameter of the Appointment. The Date field was unique in that if the date was in the past or null, I would correct it with the application of a new Date object set to today’s date.

A computer screen shot of a code

Description automatically generated

Each parameter that was not the unique, immutable ID was met with a similar mechanism to prevent null values, though in every other place it was the application of a generic placeholder value. For each Object I crafted a JUnit test script for the static testing. I created dummy versions of the Appointment for a series of controlled values with predictable outcomes and specific qualities designed to induce a failure in the meeting of parameters.

A screenshot of a computer program

Description automatically generated

These objects were created to induce obvious and measurable failures, i.e. an ID length clearly excessive of ten characters, and every parameter and criterium for its acceptability was tested. For instance, the appointmentId parameter was tested through asserting an equal value between the exact string used for the standardAppointment dummy and its getAppointmentId return value. Similar clear, short, simple tests exist for all of it, and while time consuming and repetitive to set up, they establish clear baselines for determining if there is any flaw in the program prior to release.

A screenshot of a computer program

Description automatically generated

The relevant AppointmentService is composed of a Map of String-bound Appointments, and the Strings are their keyword IDs. In the future, a singleton pattern for the Contact and Appointment Services would further secure against vulnerabilities such as unnecessary duplicity. This was not yet defined as a requirement, though. The requirements that were defined were the power of the service to add, remove, and retrieve Appointments.

A screenshot of a computer program

Description automatically generated

Another important part of designing safe programs is including feedback to ensure the user the action succeeded on the right object during runtime, so there are System.out lines printed for user feedback. Due to the limited specifications, the Tests are simply the creation of a Service, the creation of a test Appointment, the addition of this Appointment, and its removal. Note that the addition and removal of the Appointment both require retrieving it, so getAppointment is also tested.

A screenshot of a computer program

Description automatically generated

The approach to developing static tests was wholly suited to the software requirements. It was the software requirements which mandated the barebones construction upon which further development will expand; The functions work, the functions have successful tests in their most basic composition, the code is highly modularized, and each module serves as a template for future development. Each of the three core modules developed, the Task, Appointment, and Contact, has its Object, Service, and Object and Service Tests. Each of these three features were developed off the same principles of consistent and qualitative code standards and had the same technique of skeletal function-building and high-coverage testing, exactly 100% of all functional requirements are tested by our JUnit scripts. I know my JUnit tests were effective because every function that exists within the objects tested is called and measured in its output; Every necessary requirement is also tested; And when given bad input deliberately, the functions are capable of correcting it to meet specifications, from overflowing input to null input. Running JUnit also reports a 100% success rate. Just as the Appointment family operated, the Task family is composed of and tested by its requirements, featuring dummies designed to fail. This is demonstrated here:

A screenshot of a computer

Description automatically generated

**A screenshot of a computer

Description automatically generated**

1. Reflection

My responsibility to implement related feature sets and compose comprehensive tests for them meant selecting the most appropriate testing techniques. JUnit test scripts provided a grounds for static testing by iteration through a 1:1 list of the functions and parameters of each class as they were created. The static unit testing was the specific technique I put the most emphasis on throughout the development of the application, as it provided an automated frame that will benefit the entire life cycle of the project by validating historically accurate anticipated output with the function’s current output using a consistent input. Bugs and broken functions can be caught incredibly quickly with no further effort on the part of the developers furthering the application. Unit testing refers to the testing of code units, specifically modules then classes then functions as systematically organized in my practice, though a code unit can be any collective of related lines in need of testing. I tested each class and each of its subfunctions, including getters and setters. I did employ the concept of equivalence partitions for these tests, where I validated eligible input that was generic but of the true type of data that would be used in the function, where the data should transform the same way because of its validity or invalidity (in accordance with a domain of data validity). As an example, I tested with a valid number of characters in each variant of a class ID parameter, a null value, and an invalid number of characters, as with every parameter with character length restrictions. In addition to my static JUnit script based tests I employed a small degree of decision testing, which is a test type that is suited for decision-making statements. The if statements and decision makers I included in the project were data validation functions which prevent the creation of objects which have duplicate IDs into their Service maps along with invalid/null data catchers. Decision testing helped ensure the integrity of my objects and the robustness of my data.

The decision testing was part of the functions of the object which set the values of its parameters.

A screen shot of a computer code

Description automatically generated

Decision testing was used in the Service to handle various cases of data validation and, while populating the Map, instancing.

A computer screen shot of text

Description automatically generated

On the topic of unemployed tests throughout this course,

“Some techniques I did not employ were integration testing and acceptance testing. Integration testing is a test performed on how different components of an application interact. I could not test this because I have not created a main function that integrates any of the components I have developed; Everything I have produced is a limb without a stem joined to it. However, this would be an area for which I would design tests after the development reaches a point where my components are overlapping. I suppose there is a small degree of integration between any of my objects and their service, i.e. the integration between an Appointment and its AppointmentService, and there was testing performed on adding an Appointment and retrieving its information and removing it. This type of test occurred for the Contact and Task branches as well. However, I’m not confident in labeling this as integration testing because the branches’ functionalities are distinct and there is not any intercourse between any two of them. Acceptance testing is measuring the product against the functional and experiential requirements of the project. This is another testing technique which my product is not ready for, and which must wait further into development, at least until the ‘stem’ is in place that allows the application to be alive and transform data.

“Each test I did perform or chose not to perform yet was aware of has an evident practical role, but also has implications for different projects and situations. The low-level unit test was time consuming to implement and extremely repetitive to produce for each class, and ensuring its correct production meant firm code specifications had already been defined, meaning it is best suited to cases where parameter specifications are liable to be immutable. In projects where documentation is not complete in regards to parameter specifications or time is restrictive, these types of low level tests may be difficult to implement properly. Integration testing important when a project has multiple components, and more so as the number of moving parts increases. I would say all software needs acceptance testing and functionality testing to ensure the finished product works as it should prior to deployment” (Phillips, 2024, *Module 4*).

As I worked on this project, I had to take on the mindset of a lead developer as part of the ownership any good worker should express over their project, a programmer as I developed the various classes, and a tester as I ensured code integrity and quality along with writing JUnit tests, all roles adopted interchangeably. In my mindset as a lead programmer, I paid attention to the bigger picture of the application, its requirements, its inevitable use as a foundation for further development, the application’s purpose, industry standards, and the modularity of the scripts and classes. Thoroughly, I employed caution and foresight. This is what motivated the structure of the classes, their functions, and their tests, and the 1:1 function to test relationship. As an example of this caution, I did design the tests in a manner where they would be useful no matter what means of future expansion or integration befall each object. The tests are simple and straightforward; A consistent input is always ran against a consistent anticipated output.

A computer screen shot of a program

Description automatically generated

As I worked as a developer and tester, I did feel I was working against myself in a constructive way. I was flooding my boat to check for holes before I put it on the lake. I did feel a bias related to pride in my work; But I also know that if I launch my product with flaws that I could have caught, it will wound my pride more, and be costly, liable, even dangerous. I’ve always valued accountability in my work as well as quality. I also take pride in my tests, and if my tests prove flaws in my work, that is something to be proud of. If I were responsible for testing my own code in the field, I know I would be fair to my code, meaning my tests would be rigorous and honest tests of quality. While I was developing, I ran into a case where my tests once passed but suddenly started failing without changes to the object I was testing. I came back to this and completely redid the object and service classes for the module. This was important to me, so redoing them I ended up producing better tests, better classes, and ultimately better work. I would consider my approach to countering bias as *balancing* it more than limiting or negating it.

This whole semester, I had been in a series of rough positions. Because of this, I was far more tempted than usual to cut corners, and produce inferior, lower quality work. Discipline was important. I had to see the semester through with the best grades I could get. Discipline is vital in development. We assessed throughout the semester cases where failures in software engineering led to simple or even silly mistakes and oversights that led to death, trauma, or massive financial loss. Writing code and testing code requires seriousness, patience, and investment of true effort. Avoiding technical debt is important because serious work and matters are at stake, but I value the quality of my work because of my personal pride. A specific example of my commitment to quality was my revision of the Contact module. I cut corners and produced an inferior product because of my headspace. I deleted that work and completely redid it using my superior work as a reference for techniques to employ. Because of that, my product in this course was very good. It was efficient, it had very fine and appropriate tests, and I am proud of it.

**References**

Garcia, Bonie. (2017). *Mastering Software Testing with JUnit 5 : A Comprehensive, Hands-on Guide on Unit Testing Framework for Java Programming Language*. Packt Publishing.

Morgan, Peter, 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*, <https://ebookcentral-proquest-com.ezproxy.snhu.edu/lib/snhu-ebooks/detail.action?docID=5837074>.

Phillips, Justin. (2024). *Module 1: Software Testing in the SDLC*. Southern New Hampshire University. CS-320, Software Testing and Automation.

Phillips, Justin. (2024). *Module 4: Test Writing and Software Requirements*. Southern New Hampshire University. CS-320, Software Testing and Automation.

Phillips, Justin. (2024). *Module 5: Software Testing Techniques*. Southern New Hampshire University. CS-320, Software Testing and Automation.