Liam Daley

Ramsey Kraya

CS 320

Dec 8th 2023

Summary and Reflections Report

While developing the Contact, Task and Appointment software components, my testing methodology was directly informed by and aligned to the software requirements. Each test case exists either to confirm basic functionality of the system, or directly references a stated requirement. For example, the components each share at least one element with the requirement of a forced string length. This pattern exists across the software tests, where the test cases will ensure that the application properly handles input of the maximum and shortest possible length, as well as input that is too long and completely null or otherwise invalid. This pattern is used for every stated requirement, ensuring that every permutation of input that meets or fails the requirements is tested. In a simple application like Project One, this approach might seem redundant, but as projects grow in size and scope creating these kinds of tests as you go can reveal shortcomings in systems down the line, saving time when investigating later.

When performing unit tests, the “coverage percentage” is a useful tool in determining how your test cases respect the full scope of the software code base. This metric represents how many lines are exercised out of the total active lines in the solution. A low coverage percentage indicates that test cases likely miss large portions of functionality or are otherwise inadequate. I utilized the coverage percentage statistic while reviewing the test cases previously written for the project to identify any missed points. This approach did help me spot a few shortcomings in my test cases, notably a few blocks relating to exception handling that were not being exercised properly. After creating a test case to address this edge input scenario, I was able to verify that this part of the code also functioned as intended and would respond properly to input.

Through the development process, I had numerous cases where testing helped me identify issues with the software implementation. One notable example is when creating the Task service. Because the requested functionality was very similar to the Contact service, I originally used that class as a template. However, when writing test cases, I had an issue particularly with the case assessing how the class feature to update Task data entries functioned. Because I had made a mistake when creating the Task specific implementation of the system, it responded to certain inputs improperly. Requests to change certain parameters, such as a command to update the “description” field, would return an error because the method was not fully adapted for the proper inputs. After amending this error, I was able to move on to other issues and eventually verify the system functioned to requirements.

Efficiency is almost as important as accuracy and reliability in software development. It is as crucial as true bricklaying in building construction – if your code is inefficient from the start, it will pile up technical debt that must be repaid later. I found a useful indicator of efficiency while testing was how few test cases it required to exercise the entire code base. When classes properly modularize and reuse functionality, testing a single involved system will often end up testing most of the class’s functionality. As a specific example, I realized during development of the Contact class that the requirements for input parameters could be checked in specific property setter functions, instead of just the constructor. This would allow the functionality to be shared and updated between object construction and object modification, reducing redundant code and reducing failure points. After this modification, I saw the number of required test cases reduce and the overall software efficiency increase.

Through the development of the Task, Contact and Appointment objects and services, I employed functional testing methods, primarily unit testing. As each component was completed, I assembled a full test suite to operate its functionalities. This unit testing approach, intended as functional testing to specifically analyze the software’s adherence to provided requirements, involved inspecting each piece of code and determining what aspects needed a test case. For example, the Appointment service had relatively few functions. It needed to successfully add and remove classes, as well as exhibit certain behavior when an object was added with a duplicate ID, or it was instructed to remove an object with an ID that did not correspond to an entry. Using JUnit as the framework for these unit tests, cases were created to attempt operations that should return errors, and check that valid operations properly mutate the data.

Additionally, a form of Acceptance Testing was performed before project submission. This is the final stage before releasing the software to clients, and it is focused on making sure the software as it exists meets the stated requirements of the client and provides the functionalities they desire. For this project, acceptance testing (along with graded feedback) helped identify some small deficiencies in the software that could be fixed before project submission. Notably, the Appointment class had a slightly incorrect acceptance case for acceptance description input.

There is a multitude of software testing techniques, so naturally the small scale of this project caused many of them to remain unused. Some aspects of functional testing were not utilized, such as integration testing. Integration testing ensures disparate components of a system can properly communicate with each other, facilitating the operation of the complete software package. However, the systems developed in this project are (so far) disconnected, and as such make no communication between them. Each memory data object, such as the Contact, has it’s closely related ContactService, and no other related code. In a larger application, that linked Appointments to Contacts and Tasks, integration testing would be a necessity to ensure these systems work cleanly together.

Another notable field of functional testing is regression testing, which is performed after changes are made to the software to ensure all aspects still function. Because no major revisions have taken place to existing code, regression testing was not necessary. Additionally, non-functional testing was not performed at all. This category includes performance and security testing, which was not required for a project of this scale. These testing types grow in importance as project scale expands, particularly regression testing. Regression testing is made mostly automatic by the robust test frameworks developers have access to today, but it is important for tests to be added to existing classes if requirements change or previously impossible edge cases become plausible by some development in a new system. In distributed applications, high-performance environments and industries where information can be personal or confidential, non-functional testing helps ensure the software does it’s job in time and does not put anyone’s information in undue jeopardy.

As a software tester, a cautious mindset is of high importance. Underestimating code or assuming it “just works” is a dangerous approach when such assumptions can lead to shipping broken software. I experienced this first hand when creating the ContactService method to update existing objects, when I blasted through method implementation and wrote a single quick test to ensure it’s functionality. When the test failed due to a cascading bug originating from the Contact class, I had to step back and re-assess the system to make sure it met all requirements before moving on to fix the classes and expand the test cases. This illustrates how bias can negatively impact developers required to formally test their own code – because no (good mannered) developer intentionally writes poor code, we tend to assume that as long as “Hello World!” prints, the code works. I needed to train myself out of this while working on the projects, to ensure I wrote comprehensive test cases and met all stated requirements. Such corner-cutting saves you minutes or hours now, but will repay in kind down the road when a bug is discovered and the cost to fix it has increased by factors of ten. Discipline in process and rigorousness is important for testers to stay in front of technical debt, so the project team is not swallowed in bug fixing after the client reports an issue that would have been easily caught by a proper test suite.