# CS 320

# Project Two

Allen White

My unit testing approach was to digest the client requirements into functional requirements which formed the basis of testing. On top of this base (which generally included only the happy path) I would form a set of boundary conditions to further prove that the functionality created did not break down at edge cases. An example of this happy path would be the customer-specified “ The task service shall be able to add tasks with a unique ID.”, which I implemented to take a ten digit UID, check that it existed in a list, then delete the object from the in-memory storage. However, to truly test the robustness of functionality, I also added unit tests which would check if the functionality broke when a nine or eleven digit UID were supplied, or even a UID that did not exist. By doing this, we can not only ensure that the core functionality works, but that the program continues to work (fail gracefully) when dependent systems feed improper data. Based on test coverage percentage, we can see that my entire set of test classes achieve an 87.8% coverage rating. However, if we dive further into the results, we can see that we have achieve 100% test coverage except for our main() methods. It is important to remember that no test set can account for all test scenarios, so we have not truly tested 100% of possible failures, but we have achieved a level of test coverage beyond the customer and company-stated 80%.

To ensure that my code was technically sound, I generated variables that enabled repeatability and clarity for future testing. One example of this is my use of java.util.calendar and java.util.date to create test variables which set timestamps to a specified time in the past or future (eg. one second in the future or past) so that we know every time we run the test, we’re accurately testing the same boundary conditions. Secondly, if we want to adjust our boundary conditions to be tighter (eg. milliseconds in the future or past) we can do so easily, or even broaden our edge cases to ensure robust testing. Another example was an incorporation of test strings which allow for easy debugging such as “fiftyDigitString” and diffFiftyDigitString”, which again aid in checking and working with the unit test cases. To ensure that my code was efficient, I used the read out from Eclipse Junit testing results to ensure that no tests approached an unacceptable timing threshold. Each test performed in well under one second and if they had deviated significantly or resulted in higher times, I would have employed Junit’s timeout features.

During the development of the codebase for this project I used many different software testing techniques. To ensure that I designed to the customer expectations and requirements, I employed static testing through the distillation of the specifications. I performed dynamic tests of the codebase through the creation and running of unit tests in the Eclipse IDE. I performed integration testing as I developed classes which made use of other classes to create and modify groups of objects, such as when the ContactService class made calls to the Contact class. Technically, I also participated in technical reviews and acceptance testing through the iterative structure of the class, developing and submitting different application subcomponents and receiving/reading feedback. I performed regression testing as I integrated all previous classes into one project and sought out issues to resolve, ensuring that previous functionality was maintained. I also performed boundary value analysis as I made use of values inside and just outside of the normal operating range for the specifications provided.

The software testing techniques that I did not employ during this work were techniques such as system integration testing, as there was no larger system to integrate into, I didn’t perform black box testing as I was able to see the inner workings of the logic used, I did not perform branch testing, which would have tested every scenario of conditional statements in the program logic. To briefly describe the techniques above, we will look at the software development lifecycle as a guide. In the initial stages of SDLC, static testing is comprised of the pre-work before development is started, such as requirements gathering, distillation of those requirements, and design. This type of testing it the most cost effective as it can prevent countless hours of unnecessary work later on. Next, as the software product is being developed, it will go through various stages involving development, dynamic testing, and technical reviews, all of which evaluate the code and its ability to operate as expected with unit testing and outside perspective. During those phases, periodic acceptance testing will be performed with stakeholders, users, and other testers, to ensure that functionality meets expectations, especially when improper inputs (boundary conditions) are given. System integration and regression testing are performed later on as multiple functional system subcomponents, which have been developed independently are merged and previous functions are rechecked to ensure that they still perform as expected. At this time the relevance of discussing black and grey box testing (as opposed to white box testing) becomes apparent as depending on the level of transparency to the tester, the system is tested based on the output more than the internal workings of the system, black and grey a nod to the varying opaqueness of the system to the tester.

During the course of my work in this project, I adopted the mindset of a software tester, employing appropriate caution in both development and testing of the system. Caution in that our ultimate responsibility is to the end user, and we must rigorously balance risk and delivery of the core features during development and testing. I It is not enough to meet the base user requirements, we must also ensure that edge cases will not crash the system as depending on the application of the software, it may cascade into more significant impacts. When I tested to ensure that my program would handle the entry of non-existent UIDs, I made sure that no unexpected functionality would occur. Part of this testing mindset is the responsibility to restrict our own biases in development and testing. We must make sure that we push to identify creative ways of breaking the system under test as users often find creative ways of deviating from expected behavior. It would be much easier to consider only one type of user and limit ourselves to one perspective, including assuming that all users will have good intentions when interacting with the system.

We have seen through our research that when proper testing techniques are not employed, it is easy for software developers to undertest their applications, and for those applications to fail in catastrophic ways. One such example is the impact that the Boeing company’s 737 Max systems had on the lives of many passengers, pilots, and the company as a whole (Hamblen, 2020). Lack of diligence in testing, including static code analysis, can result in technical debt over time and a failure to deliver a robust, relevant product. More than simply a terrible user experience, these flaws in software product quality can result in the loss of life.

I plan to avoid these pitfalls by using a test-driven development approach, by collecting as many user inputs as possible throughout the development lifecycle including acceptance testing, and by ensuring visibility to my products by a broader team, as no single person can account for all end-user behaviors. It is easy to jump to development instead of reaffirming the underlying assumptions of a software product, to only consider the happy path for the product’s use, to not ensure regression testing of previous features, and to ship a half-finished product due to timelines. It’s my responsibility as a developer and tester to ensure that the minimal risk is taken on by my company and users when interacting with a product that I’ve worked on.

References

Hamblen, M. (2020, March). Killer software: 4 lessons from the deadly 737 MAX crashes. https://www.fierceelectronics.com/electronics/killer-software-4-lessons-from-deadly-737-max-crashes

Hambling, B. (2019). *Software Testing: An ISTQB-BCS Certified Tester Foundation guide - 4th edition* (4th edition). BCS, The Chartered Institute for IT.