Joshua Lee

CS 320

Desmond Springer

12/15/24

My testing approach for this project was diverse depending on the class I was working on. The Contact class required string variables including contactID, firstName, lastName, phone, and address. Each string variable couldn’t exceed a particular character count or be null. On creation, all of these variables are checked and an exception is thrown if an error is found. For testing I test proper creation and that all of the variables are accessible and correct and I also test improper values to see if an exception gets thrown. For the improper input tests like testInvalidFirstName it needs a (expected = IllegalArgumentException.class) leading the function to make sure it expects an error as a result. I tested all of the proper values and the improper ones reaching all of the edges of the variables for testing. Between the tests testContactCreation, testInvalidContactID, testInvalidFirstName, testInvalidLastName, testInvalidPhone, and testInvalidAddress we ensure that improper values throw exceptions on creation and proper values work properly. For the appointment class, there are two string variables with character length maximums that cannot be null and a date that cannot be in the past or null. Upon construction, all of these variables are checked and an exception is thrown if an error is found. For the JUnit testing we see the test cases testValidAppointment, testInvalidAppointmentId\_null, testInvalidAppointmentId\_TooLong, testInvalidAppointmentDate\_null, testInvalidAppointmentDate\_PastDate, testInvalidDescription\_null, and testInvalidDescription\_TooLong where we test all the edge cases for each variable and ensure exceptions are thrown for the invalid instances. This was the minimum amount of test cases to ensure the quality of the program testing all of the improper values and proper creation of the Appointment. Lastly, the Task class is all string variables similar to the Contact class and the test cases also just insure proper construction doesn’t throw an exception and yields the proper Task, and improper values throw exceptions.

Reflecting on the development of this project some testing techniques used include Boundary testing like in the Contact class ensuring that a phone number is exactly 10 characters. Any more or any less the test throws an exception. This is also Negative testing where you make sure that improper values are handled gracefully and don’t continue into the system. We see this a lot when the test cases are led with expected = IllegalArgumentException.class in parentheses before any of the test cases for each of these three classes. Negative testing is of the utmost importance because bogus values come into play when using any system and you want to be sure your system handles them when they happen. Some methods I didn’t use include parameterized testing which would have been helpful for redundancy throughout all of the testing. I could have parameterized the construction of proper values for testing service classes whereas I constructed an individual object for each test class. This just increases redundancy and is very helpful for large files. I also didn’t do any exploratory testing as the project was rather streamlined and there wasn’t much of a system to test and go over. If there was a full system involved I could have done performance testing and regression testing for new versions of the system but this was more of an initial interface.

The mindset for this project definitely had me employ caution because you have to make sure the program is sound before it is submitted just like if you were to merge code to the main branch of a big project. Everything needs to be tested to make sure it is complete before submission otherwise you merge to the main branch with a bug or there is a problem with the submission of work. For this project, we were given requirements that are able to be put in black and white and they can be directly tested with JUnit testing ensuring the completeness of the program. Given the requirements, I can limit bias in my review of the code so long as each requirement is tested directly which is what I’ve done for the Appointment, Contact, and Task classes. Commitment to quality is something that pays dividends over time if practiced. Alternatively cutting corners and skipping proper testing can lead to costly bugs or confusing results. Say for example we let someone make an appointment in the past and didn’t test we may in the future implement paying in advance for an appointment and if they pay for an appointment and they accidentally schedule it for a year in the past you’re left with a time-consuming problem that you could have easily avoided with proper testing. Code documentation, rigorous testing standards, and continuous refactoring are good ways to avoid Technical Debt, ensure maintainability, and build trust with users and the product owner.