Southern New Hampshire University

CS-320 Project Two

By Paul Schwartz

Summary

The approach aligned with the software requirements by reading over requri9metns and making sure each of the requirements are included and tested. With there being three series being created, Contact Service, Task Service, and Appointment Service each one had its own requirements and testing that was required. With each milestone of the coding part of the project listing out those requirements and then using coverage percentage to make sure that everything was covered under out testing portion of the code.

Contact Class had requirements for its different objects such as contact ID not being longer than 10 characters, first Name, last name and the phone strings not being longer than 10 as well. Also, requiring an address field that had a 30-character limit. Then at the same time making sure each object could not be null or empty as well. Using code like **if** (contactID == **null** || contactID.length()>10 ) {**throw** **new** IllegalArgumentException("ID is Invalid");. This set up the main constructor to allow creation of those objects while settings limitation for them.

Since each Service had its own sets off requirements each one had to meet using similar code above, then having to move on to creating junit tests for each object. With the example above you would not just test one part of it, it would need to be testing for a null/empty entry and over 10 characters. This one object would need at least two junit tests to coverage it correctly. Those tests looked like **void** ContactfirstNameToLong() { Assertions.*assertThrows* (IllegalArgumentException.**class**, () -> {**new** Contact("12345", "12345678910", "12345", "12345", "12345");}); and **void** ContactIDNull() {Assertions.*assertThrows* (IllegalArgumentException.**class**, () -> {**new** Contact(**null**, "12345", "12345", "12345", "12345");}); This is testing both over 10 characters and a null/empty entry.

This showed how my code was technically sound. Not only the tests passed and worked correctly but the creating of those objects were successful. Showing no errors and issues when it was compiled. Using that and using the class resources allowed me to correctly create arrays and constructors for each object while settings the limitation for those entries. Using the arrays and objects correctly ensured correct code usage and that that code was efficient. Efficient in the way that where was no unnecessary tests or unnecessary code used to bloat the Services created and it’s stuck to the requirements of the program.

Reflection

The main testing techniques used in the projects where black and white box testing. These testing techniques use specification or structure in how the test, such as yes or no it did not work. Using this technique worked well with the project specifications since it had specification things the system should do or not do. Basically, having tests for invalid or valid inputs with having set boundaries for those entries, like not having more than 10 characters. Using all those requirements and testing techniques to break down the programs into smaller sperate parts to make sure everything is covered, and the coverage percentage is at an acceptable level.

Since being new to junit testing a technique that was not used was experienced based techniques. Experienced based techniques are used when the end user and testers use and test the program. This would require users and testers with experience in using that type of software and finding errors. This is almost like a code peer review but where they use the software for its intended purpose. Experienced based techniques are also a great way to find if the code is efficient and works as intended because of past experiences there might be better and easier ways to do certain things depending on the programing language being used.

Each technique has its strengths and weaknesses, I can see using white/black box testing for more simple instructions within programs such as certain data entries. Whereas the experienced based might be better used with programs having a lot of moving parts or not clearly defined requirements. Using both types may have a better coverage of issues or problems that could show themselves.

I used a lot of caution, I usually worked on the project in small parts to make sure they worked first before moving on to a lot of different things. Like how each milestone had the project in three different parts. Was not trying to code all the services at once, with Contact Service working and testing as intended this allowed a good foundation for the next two and built confidence in doing them.

Bias could be a major issue if I was a more experienced programmer, I have seen bias like this in the construction industry with someone with tons of experience is not willing to learn new ways or won’t use more efficient ways to do things. Why I would personal use code peer review a lot, that way someone can trust but verify my tests and code work as intended.

The more programming and projects I do the better I understand how important quality is in programming. It tends to work like Jenga, if the foundation is not very good its easier for everything to fall apart and stop working at the top. This would require more work to redo things to work correctly and having to refractor a lot of code because of early mistakes. To avoid technical debt, I would create and do testing early and often to ensure a great foundation. While also having open communication with the client on the software development and requirements to allow changes or adjustments as necessary.