Project Two

Steven Anderson

1. Describe your unit testing approach for each of the three features.

When making my approach to the requirements I looked at what each variable could and couldn’t be. Such as in Contact the ID, firstName, and lastName variables couldn’t be more than 10 characters and couldn’t be null. The phone had to be exactly 10 characters and couldn’t be null and the address couldn’t be more than 30 and couldn’t be null.

To make sure the inputs matched all these restrictions I had in the main contact.java file where the inputs were assigned if statements that would throw exceptions if the variable inputs failed requirements. With ID, firstName and lastName being tested to see if they were either null or >10, if so than throw the exception. With phone seeing if it was null or length !=10. And address to see if it was null or >30. I also added these into the setters to make sure that if a contact was modified or added it still matched the requirements.

When I went to JUnit test these, I made variable inputs that both matched and didn’t match the requirements. That way I could write JUnit tests that could use these variables to test for what was needed. Like testing to see if any value exceeded the character limit or if any value was set to null. In my tests I used the proper input of (123456789, Luke, Skywalker, 4876952463, 5689 Ben Farm Tatooine). This input matched all requirements for length and non-null variables. I than made “bad” variables that were too long for the requirements to put into my JUnit testing. This way I could know that any variable thrown through that didn’t fit the requirements would fail out.

Than to test my ContactService.java file which included my updates, add and delete options I wrote additional contact variables that matched to requirements for testing. When doing my deleteContact test I used all three by adding each to the contact array list and then calling on deleteContact to delete one with a positive ID and making calls with false IDs to see if it would catch the false IDs. And for updating I actually used 4 different tests for the four variables other than ID. With a test just for updating the first name, a test for the last name, a test for the phone and a test for the address. I also included a test for adding a new contact and used my premade valid variables to see if the system reacted right. I repeated this kind of testing for both the Appointment files and the Task files with changes based off their own unique requirements in regard to length or updating.

1. Describe your experience writing the JUnit tests.

To test and make sure my code was sound I used JUnit testing to catch exceptions and verify features worked properly. In my ContactServiceTest.java file that stored the JUnit testing for my ContactService.java file I made sure to check possible scenarios for variable inputs. Such as in my deleteContactTest I used the example contact ("457821547", "Uncle", "Owen", "1259876534", "5689 Owen Farm Tatooine"). I than ran 3 different tests to see if the system would catch false ID’s and then finally to see if it could find the correct ID.

//test to make sure it returns false

assertEquals(false, temp.deleteContact("457628547"));

assertEquals(false, temp.deleteContact("457821714"));

//test to make sure it is true

assertEquals(true, temp.deleteContact("457821547"));

In the example code I have above you can see the two tests being ran without a valid ID to match the Uncle Owen ID number, these test both returned the false for failing to find the correct ID. Then I ran the test with the correct ID which returned true because it was able to find and delete the ID matched contact. I also used Array Lists to make sure new contacts or appointments or tasks were added to the list for ease of access and for proper coding practice. Without the array lists each new contact would either have replaced the last or would have had to get its own variable addition which would bog down the code and possibly lead to memory leaking and critical errors.

To make sure my code was efficient I went through each section and made sure no redundant coding was being called for no reason, and no non used code was kept in the program. Such as when I wrote my setters for Appointment, Contact and Task I didn’t add setters for the IDs. This is because all of them had the requirement that ID be unable to be updated. Such as in my Task.java file I only wrote the setters for name and description.

//set name

protected boolean setName(String name) {

if (name == null || name.length() > 20 ) {

return false;

}

this.name = name;

return true;

}

//set description

protected boolean setDescription(String description) {

if (description == null || description.length() > 50) {

return false;

}

this.description = description;

return true;

}

I also made sure that when writing my code each section was organized by purpose. Like having my setters together or my getters together. I did the same thing in the test.java files where I kept the test organized by what they did, with all the update tests being grouped together for ease of reading.

1. Testing Techniques

When testing my code, I used a lot of techniques. I used the JUnit testing to make sure every decision the code made was checked. From verifying a variable matched requirements to making sure adding delete and updating would both work or reject based variable acceptance. I also hand wrote out the logic the program used from when a variable was submitted to when it was either added or updated. That way I knew where testing needed to occur based off each step, a form of white box testing. I would make sure I understood what each part of the code both needed to do and how it needed to do its function. I would write for testing both variables I knew would pass the initial requirement tests and would fail them. This would also be an example of unit testing. Before writing the JUnit tests I would actually write the tests within the java file first to make sure the file ran with everything working. By adding the variables and having them print out within the file.

I also used acceptance testing when I testing to make sure each variable was being processed right. Initially I didn’t do well in the first assignment since I didn’t think about how the variables were being input and didn’t write the if statement tests right, also not adding them to the setters was a major mistake. By the time I got to the 3 assignment and project 1 I had corrected this and gone back to fix this in the first two assignments so they passed all acceptance testing. I also used Integration testing when working with my searches and my cross-file accesses. Ensuring that when a test file was written it called on the .java file it was testing. And the search functions were communicating well with the add, delete and update functions which would call on the appointment.java, contact.java or task.java files. I also did a lot of Regression testing since I was making a lot of changes as a went. This included retesting after writing every JUnit test to make sure they passed or retesting when I made changes to my search functions.

For this project I had no use for quite a few of the testing techniques. Such as performance testing which would have been used if I need to see the speed and stability of the program. Since this wasn’t a requirement, I didn’t test for this. And usability testing, which would have been testing for how the user experience would be. Since all my inputs were entered by test and non by a user there was no need to see the user experience. I also didn’t test for any non-functional testing. This would have been testing to see the performance, and usability as a whole and since that wasn’t really needed without a required performance rate or user functionality, I didn’t use testing for this.

With each of the techniques they all have their specific purpose in practical use. Such as acceptance testing which for practical use would be best for making sure variables are being used by the system and correctly. Integration testing is used practically by companies to make sure when multiple people are working on a project their code doesn’t clash and actually works together with no redundancies or rejections. With regression testing it is best practically used when rewrites are made during a project to make sure the rewrites are performing the way they need to and no problems or errors have been caused by the rewrites.

1. Mindset

I adopted a very rigid mindset when writing this code, not looking at the whole code until I knew each of the blocks were completed correctly. I thought of it as a complicated puzzle in which each piece had to do a function to work. Focusing on each piece then adding them slowly to make sure they worked with the other piece and fit properly. Like when writing my JUnit tests, I would write each test and then run the JUnit testing until that test worked before moving on to writing the next test. That way I knew that piece was completed.

I tried to limit by own bias when writing by taking time away from it and coming back. This would sometimes help me with resetting my perspective on my code. I also would copy the code out of eclipse into word documents and highlight sections I was worried about and try and rewrite it in a few ways to see which way worked best for me. When a software developer is testing their own code with bias it can cause problems by making them think like a personal echo chamber. Especially if they took a while to write the code, they will be less likely to admit when it could either be written better or when their logic is flawed.

Quality is key in writing software. Without focus on quality coding you could end up with programs that fail when deployed, leading to wasted time and money on projects. You could also end up with incomplete projects from cutting corners to the point that if the program needs to be updated it end up having to be rewritten due to the code being so poorly written. It can also lead to devastating security flaws if that isn’t testing for. Leading to breaches in security and either privacy concerns or sensitive information leaking. To avoid this myself I plan on relying on those on my team to best help me cover my blind spots while also helping to cover their blind spots. Ask as often as I need to for people to run tests on my code or read it and see if I make a mistake. Also, I like to write out logic charts and flow charts to visualize my program running from start to finish. This practice can help me keep a focus on the logic of my program and make sure it doesn’t have features left untested since I can make sure I test at every logic decision or branch in the chart.