CS320

Project 2

Summary and Reflection

Kevin Jordan

I followed a consistent unit testing approach using JUnit. Each test class was written to confirm core functions from the requirements. This included inserting, updating, and deleting data from datasets. I also performed validation tests for things like string length, null inputs, and other more specific conditions like date values being accurate and not showing past values for appointments.

Aligning with requirements was my main goal with testing as it allows us to confirm certain operations are functional. I have a 2 monitor setup so I would keep a list of requirements handy at all times on the second screen to allow me to adhere strictly to them. One example is in AppointmentTest, I verified that an exception is thrown if the description exceeds 50 characters. I did the same for if appointment dates are in the past. Another example is in ContactTest where I confirmed that null or names that are too long also result in test failure. These were specifically to adhere to the requirements for each class.

To verify the effectiveness of the tests I made sure there was proper code coverage to ensure I was testing as much as possible, and made sure individual tests were requirement specific and clearly labeled. This way we can run through the requirements like a checklist and also remain modular when fixing tests. I ensured that each method’s logic (including failures) was executed properly. An example would be in AppointmentServiceTest, I tested valid and invalid appointment additions, and confirmed that the correct exceptions were being triggered. To ensure effectiveness further I also tested edge condition requirements. For instance if an ID had 10 characters versus if it had 11 characters to validate correctness.

I ensured it was technically sound in a few ways. First, was following naming conventions and keeping each test name descriptive and clear as to what its function is. Examples include testAddValidContact() or testDeleteNonexistentAppointmentThrowsException(), which are both very clear as to what their test is for, without much explanation needed which makes structuring the code clean. I applied consistent logic across tests as well as used purposeful assertions to verify the code was behaving as expected. An example would be in ContactServiceTest, I verified that a valid contact could be added and retrieved correctly.

***Contact contact = new Contact("1234567890", "John", "Doe", "123 Main St", "5551234567");***

***service.addContact(contact);***

***assertEquals("John", service.getContact("1234567890").getFirstName());***

This test confirms if it is successfully added and a proper retrieval. I also avoided duplication by defining constants for the test values in the file, as well as grouped similar tests together for organizational reasons.

This goes hand in hand with efficiency. I maintained efficiency by attempting to remove any redundancy using helper methods and constant values. I received some great feedback in this regard where I was using redundant tests and adjusted accordingly. Avoiding duplicate tests by storing them at the top of the class. I also made sure that the tests were not bloated and made sure each method only asserts one logical condition. I did this so failure would be much easier to isolate and identify. An example would be in TaskServiceTest I tested invalid inputs using the minimum required information to reproduce the problem, instead of recreating a full task each time. This makes it execute faster and is much cleaner. Only initializing what is needed for the specific test requirement.

***Task task = new Task("1234567890", "Clean", "Clean the house");***

***taskService.addTask(task);***

***taskService.updateTask("1234567890", "Mop", "Mop the floors");***

***assertEquals("Mop", taskService.getTask("1234567890").getName());***

In separate tests I reused the same ID to avoid repeating object creation when not necessary and just handle the exception.

***assertThrows(IllegalArgumentException.class, () -> {***

***taskService.updateTask("1234567890", null, "Missing name");***

***});***

This makes the tests cleaner since they do the minimum to assert behavior it makes it more readable and execute faster.

The testing techniques employed involved unit testing with tightly focused methods and functions. I used boundary testing to check length limits for inputs as well as verifying date constraints (dates in the past). I also did exception testing by making sure invalid inputs trigger the right exceptions by using assertThrows. I tested for expected valid inputs to ensure the system functions correctly when used properly. I also used code coverage to make sure a minimum amount of my logic was being tested. This helps guide where new tests may be needed, and that everything is being triggered correctly.

There are many techniques used to test code and some were not used in this project. I did not use integration testing. Our application was not directly interfacing with a database so this technique would not apply. If the software ended up expanding to store information in a database we could use integration testing to make sure our data is getting where it’s supposed to.

Unit testing helps for early bug detection, as well as allows for quick iteration. These tests help spot where logic is failing and help prevent regression. For example, we used unit testing to confirm core functions like addContact(), updateTask(), and deleteAppointment() while isolated. Boundary and exception testing help us ensure that user inputs are valid. Our projects had requirements for character length and boundary tests were required to ensure it was adhered to. We used exception testing by creating assertThrows to confirm invalid inputs. I.E. Null, too long, date in the past. We also use positive testing to make sure the main functions actually…function.

***Task task = new Task("1234567890", "Clean", "Clean the house");***

***taskService.addTask(task);***

***assertEquals("Clean", taskService.getTask("1234567890").getName());***

This shows a properly structured task can be stored and retrieved.

As a fairly green computer science student caution is very important for me when working on software development projects. I had a few bumps in my projects but the feedback given by my instructor was very useful and I really learned a lot. I learned to exercise more caution as I went as any small change in logic can cause a major change in the rest of the code and problem solving for this can be hard. This caused me to adopt a very methodical step by step approach where I double checked each test to ensure it passes and or fails in each expected scenario. If you are not cautious you may spend more time troubleshooting and starting over than accomplishing your goals. One example is in AppointmentServiceTest I was running into errors running tests and could not figure out why. There a system time issue where the date being input was in the past. I had to update it to verify past dates get rejected to fix the issue. A little caution could have prevented this and had it ironed out before the errors.

Limiting bias is something I am always focusing on as I am still very green in the software development world. One way I limited bias was by removing ego and carefully addressing instructor feedback each week. This allowed me to make my code more efficient as I had redundant tests running when I didn’t need to and this helped me to address that. I also try to test for edge case scenarios to try to limit bias. Like in ContactService I had forgot to check for null addresses in the Contact constructor. By reviewing the code without bias I noticed this and added a specific exception test for the situation.

***assertThrows(IllegalArgumentException.class, () -> {***

***new Contact("1234567890", "John", "Doe", null, "5551234567");***

***});***

Discipline is about understanding good practices and sticking to them. Make sure names are clear and organized. Make sure your structure remains consistent. Make sure we are achieving appropriate coverage. Sometimes things work anyway and it’s important to stay disciplined so you are prepared for the situations it doesn’t work, and avoid creating unnecessary confusion. Cutting corners will lead to issues that potentially compound on themselves in the future. It is better to take the time now as opposed to try to undo a major problem later down the road.

I will avoid technical debt in the future by making sure I write tests immediately after implementing features, even edge cases. I will rely on the tools and techniques I learned in this course like unit testing, exceptions, and code coverage. I will also make sure that I stick to best practices with clear naming, structure, and organization. I find that this is one of the biggest things you can do to improve and makes troubleshooting and expansion a million times easier. I also will make sure to avoid the quick easy way out and make sure not to cut corners.

References

Hambling, B., Morgan, P., Samaroo, A., Thompson, G., & Williams, P. (Eds.). (2015). *Software testing: An ISTQB-BCS certified tester foundation guide* (4th ed.). BCS, The Chartered Institute for IT.

TutorialsPoint. (n.d.). *SDLC - Overview*. TutorialsPoint. <https://www.tutorialspoint.com/sdlc/sdlc_overview.htm>

García, B. (2017). *Mastering software testing with JUnit 5: Comprehensive guide to develop high quality Java applications*. Packt Publishing.

TutorialsPoint. (n.d.). *JUnit - Test Framework*. TutorialsPoint. <https://www.tutorialspoint.com/junit/junit_test_framework.htm>

Parasoft. (n.d.). *JUnit tutorial: Setting up, writing, and running Java unit tests*. Parasoft. <https://www.parasoft.com/blog/junit-tutorial-setting-up-writing-and-running-java-unit-tests/>