Grand Strands Systems Summary and Reflections Report

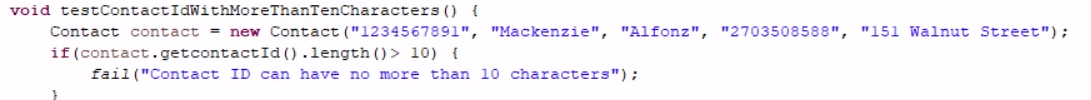
CS320 Software Test, Automation

Nicholle Caudy

8/12/24

**Summary**

The software requirements indicate fields that could not be null or have over a specified number of characters. The contact class specified Contact ID (10 chars), first name (10 chars), last name (10 chars), phone (10 chars), and address fields (30 chars). The Task class specified task ID (10 chars), name (20 chars), and description (50 chars). The Appointment class specified Appointment ID (10 chars), Appointment Date field that could not be in the past, and Appointment description (50 chars). The service classes indicated what fields should be updatable and they needed to be able to add and delete by the unique IDs. Here are examples from the contact test class, and the task service class.



A close-up of a computer code

Description automatically generated

A computer code with text

Description automatically generated

I worked on covering all aspects of my code to ensure that everything was being tested appropriately. With an acceptable coverage percentage of 80%, I aimed to have a coverage percentage rate of 90%. I was able to achieve this with areas that need improvement. I would code one test at a time and run the test to ensure it ran correctly before continuing to the next test. This helped avoid long error corrections and allowed me to make adjustments in both the Service classes and Test Service classes.

A screenshot of a computer

Description automatically generated

To ensure my code was technically sound I used the variables that needed to be added throughout the code and used the Illegal Argument Exception to catch errors for null values and to ensure the number of characters specified for the project were not exceeded. I also used getters and setters to set the values for each variable. For example, lines 43-50 in the Contact Service class, 26-28 in the Task Service class, and 12-14 in the Appointment Service class.

public void addAppt (Date apptDate, String apptDesc) {

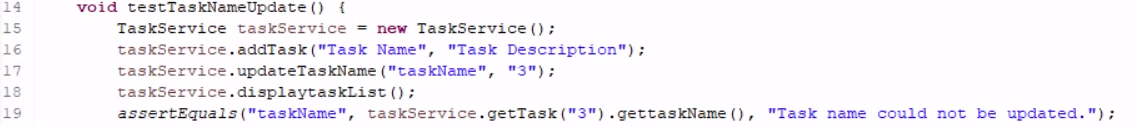
Appointment appointment = new Appointment (apptDate, apptDesc);

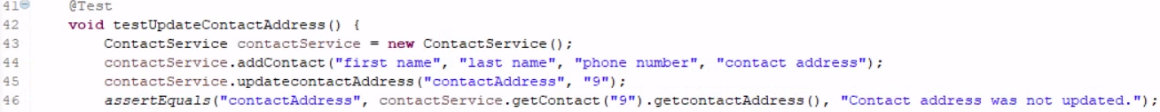
apptList.add(appointment);

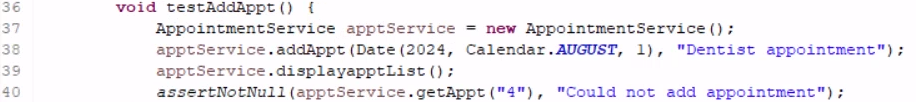
A computer code with text

Description automatically generated with medium confidence

To ensure my code was efficient I avoided using redundant code when I could use one test to cover multiple areas. This would include null arguments, and the length of characters used. This kept runtime short and efficient. I used reusable components whenever possible. I also used easy-to-remember variables to avoid confusion and make for easier readability. I used exception handling that also correlated to the JUnit testing process.







**Reflection**

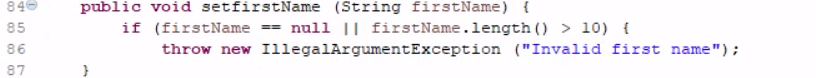
The Contact Service milestone required a unique contact ID, first and last name, phone number, and address fields. All but the contact ID field needed to be updateable, not null, and allow contacts to be added and deleted using the ID. The Task Service milestone required a unique ID, task name, and description. The task name and description needed to be updateable, not null, and they needed to be added and deleted using the ID. The Appointment Service milestone required a unique ID, date (that could not be in the past), and description. We needed to be able to add and delete appointments using the ID. All fields had a set number of characters that could not be exceeded. The software testing technique performed was unit testing which is a white box test. Unit testing is used to test for risk reduction, verify the code has the functional and non-functional behaviors specified by the requirements, and find any defects in the unit. Specifically, JUnit in Eclipse was used. JUnit is used to write and run repeatable tests. I used assertions to throw exceptions to ensure that my code stayed within the requirement parameters. For example, if the task name is null the test would throw an exception with an invalid task name message. Manual testing was also used as I read through the code to look for errors and I used the tools available in Eclipse to find errors and correct them early.

Unit testing is only one technique when testing code. The software testing techniques not used for the Contact Service, Task Service, and Appointment Service milestones are integration, system, and acceptance testing. Integration testing ensures all the units work together because even if the individual units work independently, they may not function properly once integrated. If we integrated all the milestones, we would need to run an integration test to ensure all the units worked together as a mobile application. This would test the functional and non-functional behaviors, find defects, and reduce risk. System testing ensures the functionality of the project from an end-to-end perspective to ensure the system can handle real-world conditions using various scenarios and edge cases. This testing process looks for stress handling, reliability, performance, maintainability, recovery, and usability. Acceptance testing ensures that the software system meets all the user requirements and expectations and is done independently of all the other tests.

JUnit testing was used for its practical use of writing and running repeatable automated tests. These unit tests were used to test the code for errors and to ensure it functions according to the requirements provided. If we integrated all three milestones, we could run an integration test to see if it runs together and functions according to the requirements. System testing is used to ensure the software system can handle real-world conditions. This could be used for video game development, to test gameplay integrity no matter the number of players. Acceptance testing is done independently of the other tests and ensures the software system meets all the user requirements. A cell phone banking application acceptance testing would need to ensure that it allows the customer to securely see their banking account information, move funds, find branch locations, update information, see transactions, and have functional links to the bank’s website. This would also test user-friendly functionality, security, maintainability, and refresh rates.

**Mindset**

The mindset that I adopted while working on this project was to keep it as simple as possible and make all code testable. I wanted to code in a way that I could create a runnable test to ensure that the code was technically sound and would work appropriately. I considered my large lack of experience so I would code one small piece at a time and then write a corresponding test. I tried to ensure that my tests covered as much as possible. For example, I would test the character restrictions, the fields were not null, and the different aspects of the application were able to add, delete, and update as specified in the requirements.



A close up of text

Description automatically generated

Eliminating bias can be difficult especially when you are responsible for coding and testing. I removed some of that bias by simply following the requirements and testing based on those requirements instead of my code. This allowed me to ensure that I was meeting the requirements and removed my bias by focusing on the assignment as opposed to my code alone.

A screenshot of a computer

Description automatically generated

There were times when I would run a test, and it failed or there were errors, I would get frustrated because I just knew it should work. I would have to walk away and come back to figure out where the issue was and correct it. I also think it is important to work as a team to help remove any bias, a coworker may find an error that I may have missed or find a more efficient way for the code to run.

Quality as a software engineering professional is of the utmost importance, testing the quality is a vital part of the SDLC. Small errors such as a misplaced decimal point can cause a catastrophic failure and something like this can be caught in the testing process. I plan on testing often because it is essential to avoid technical debt, this allows us to catch issues early so they can be corrected without causing mass amounts of rework. I plan to use the knowledge of those around me to help make sure I create and properly test for quality, efficient, and safe code. We all must remember that we are human and accept that errors do happen even if we have huge amounts of experience. We should never cut corners when writing or testing code because a faulty program can cause large monetary losses for the company and its clients, leak vital personal information, and worst-case cause injury and or death if something is missed and causes a catastrophic failure.