CS-320 Project Two

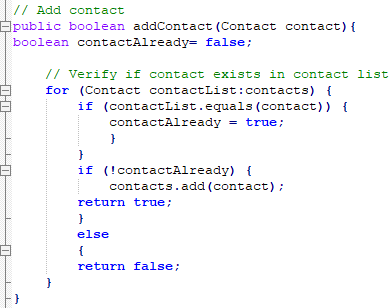
Colton Stiff

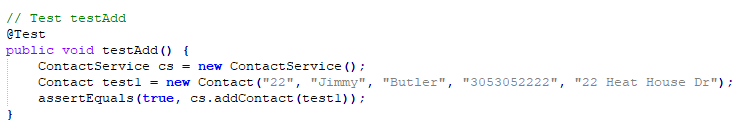
08-15-2021

Southern New Hampshire University

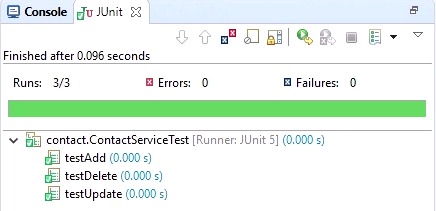
The objective of this project was to develop, test, and deliver a mobile application comprised of three services that enabled users to add, update, and delete contact, task, and appointment objects within the application. To ensure that the delivered product thoroughly satisfied the customer’s acceptance criteria and functional requirement, I took an overall methodical testing approach to each object and its corresponding service. I found that the more meticulous and consistent I was with developing the code for each object and service directly correlated to the accuracy of the testing results.

I chose to look at each milestone's development and testing phases from a holistic point of view. Instead of developing each object or service class first, then creating their respective tests class and performing the JUnit tests afterward, I worked on the code for the object or service class and their corresponding test class at the same time. I created and structured my code one section at a time to test each requirement before moving to the next one. I found this continuous iterative and “round-robin” approach to work on each class per testing requirement was not only easier for me to develop and test in a more organized manner, but it most likely also save me time when it came to troubleshooting and refining parts of code since I was working on it in smaller chunks. This also worked quite nicely when performing JUnit tests in the IDE because the test results are displayed in an itemized manner according to each test. The screenshot below is of the code I developed for my ContactService.java class, which shows the addContact function. Since the requirements of the ContactService.java class state that to add a new contact that it must not already exist, the program uses a for loop to determine if the contact already exists in the contact list:

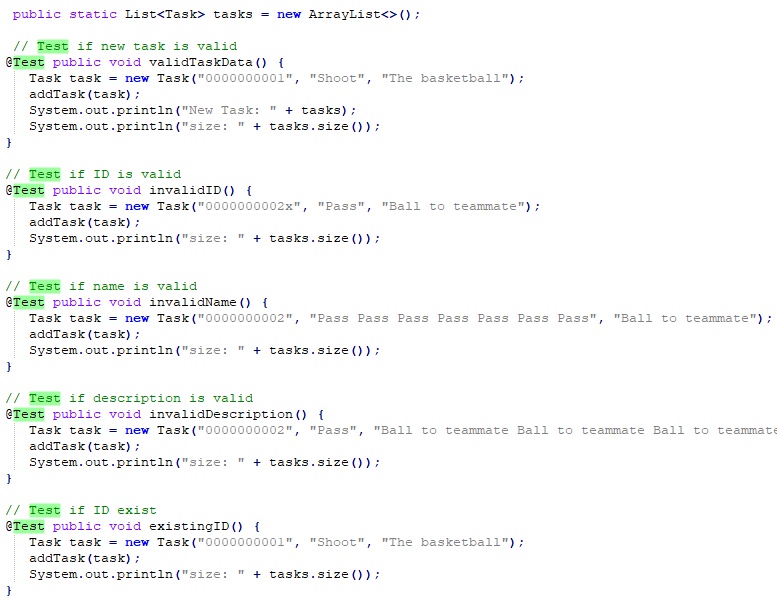


The ContactServiceTest.java class then performs this test by attempting to add a new contact (one that doesn’t already exist within the contact list) by adding a new contact ID, first name, last name, phone number, and address:

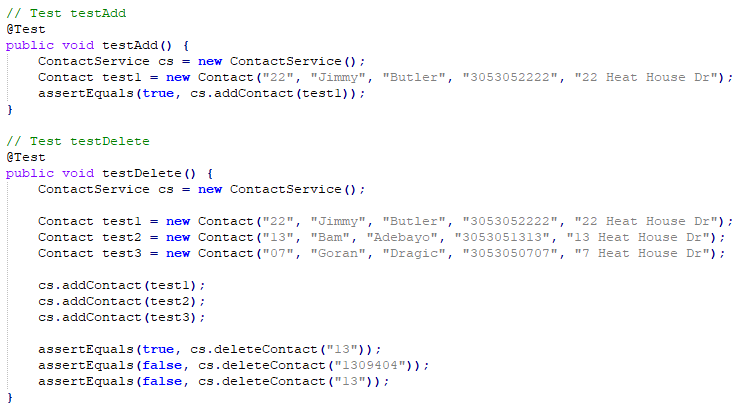
Since the new contact being added is new and doesn’t already exist in the contact list, the JUnit test successfully passed:



I continued this process to work on each test individually until I received a successful JUnit test before moving on to the next. The extent of this approach not only aligned to the software requirements and satisfied them thoroughly, but it also contributed to the effectiveness of the test but was also visually pleasing when assuring that 100% test coverage was achieved.

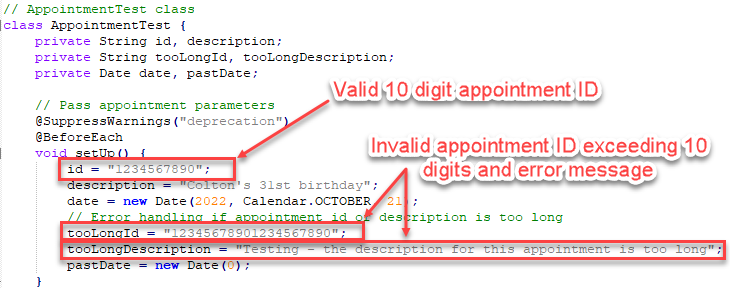
While writing the JUnit tests for each of the three features, I ensured that my code was technically sound by confirming that it was free of any syntax, compilation, or runtime errors. I confirmed that the organizing, spacing, and syntactical accuracy of my code by ensuring that there weren’t any instances of misspelled variables or statements, incorrectly used comments that weren’t commented out, missing quotation marks for string values, or misuse of parenthesis or brackets. For example, the code from my TaskServiceTest.java classes.

The code’s accuracy can also be proved because the program runs free of any syntax, compilation, or runtime-related programming errors. I was also able to prove that my code was efficient because I used concise and consistent structuring throughout all object and service classes and their respective test classes. For example, the snippet of code from my AppointServiceTest.java class is organized similarly to the previously referenced TaskServiceTest.java class:



Despite having two separate service classes, the TaskServiceTest.java and AppointServiceTest.java classes use the same formatting, spacing, and naming conventions. This proves that the programs are also free of conciseness or logical programming errors and validates their efficiency.

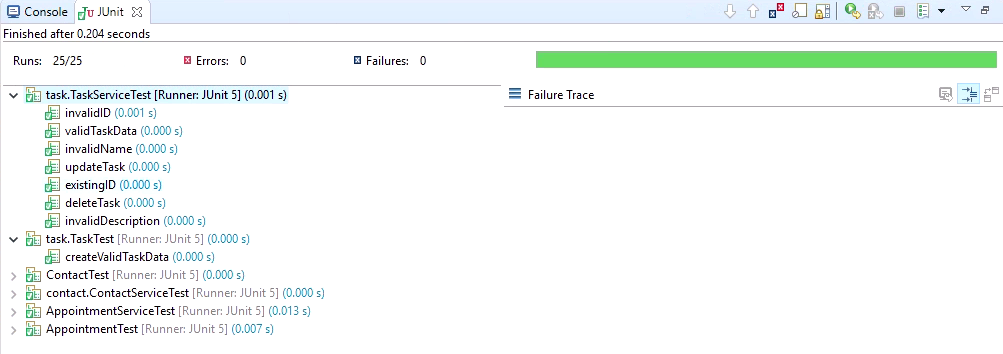
The software testing techniques that I employed for this project revolved around testing the specifications or the structure of the code that I was tasked to develop and test for object, service, and test class, which would be considered examples of black-box and white-box testing. Since I was provided the specifications of the java classes that I was tasked to develop and test in each milestone but didn’t know anything about the model these classes may belong to, I was performing black-box testing. The specification-based techniques that I employed were boundary value analysis, decision table testing, and equivalence partitioning. For example, the Appointment.java class in module 5 has a specification requirement that “the appointment object shall have a required unique appointment ID string that cannot be longer than 10 characters, and I was tasked to test both a valid and invalid input, which would be an example of equivalence partitioning testing. I performed equivalence partitioning in my AppointmentTest.java class, where I tested an appointment ID that is valid and doesn’t exceed 10 characters (see snippet below):



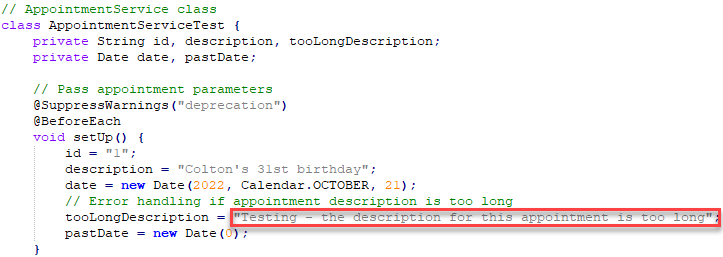
This example also depicts the boundary value techniques I employed because I tested the error handling of an ID that exceeds 10 characters and fell outside of the specified boundary.

The other software testing techniques that I didn’t use for this milestone revolved around white-box testing or structure-based techniques, such as error guessing, exploratory, and use-case testing. Since I didn’t have any prior experience as a non-biased tester, I couldn’t employ error guessing or exploratory testing techniques as both would require familiarity with the design and structure of the tested system. Similarly, I couldn’t employ use-case testing techniques as I wasn’t aware of any relevant business-specific requirements or testing scenarios. For these reasons, I couldn’t leverage many testing techniques involving interactions between “actors” and the system as I was the only user involved and had no familiarity with the system being tested.

The black-box and white-box testing techniques I discussed have practical uses and implications for different software development projects and situations. The key difference between the two testing methods is that in black-box testing, the tester knows what to test but doesn’t necessarily know the reasons why, while in white-box testing, the tester typically knows both. More specifically, black-box testing involves specification-based testing techniques. It’s intended for a non-biased tester that doesn’t know the internal structure or design of an implementation. In contrast, white-box testing revolves around structure-based techniques where the tester is familiar with the structure and design of the implementation they’re testing. Specification-based techniques related to black-box testing would be practical in scenarios where the testers are unfamiliar with a particular system to ensure that the results or non-biased, such as a random number generator. Alternatively, the structure-based techniques related to white-box testing would be most practical in scenarios where the tester should be aware of design or structure of a system, such as a Physician and electronic health record management system (EHMR).

While working on this project, the mindset that I adopted was that of a software tester who employed caution, patience, and unbiased expectations, especially when testing the code. Since the software testing techniques were often examples of black-box testing, the code's business requirements, design, and structure were not things that I was aware of. For these reasons, I needed to act with caution and appreciate the complexity and interrelationships of the code I was testing. As mentioned previously, I examined and tested each section of code per the specific testing that was being performed. 

For example, the TaskService.java class requirements that were being tested in the TaskServiceTest.java class were to confirm whether the task ID, task data, and name were all valid concerning the object class requirements, but also if the add, delete, and update functions were valid for the service class requirements.

The ways I tried to limit my biases when reviewing the code were not too complicating. When testing, I prioritized consistency and conciseness over my actual understanding, expectations, or familiarity with the code. This was a terrific exercise to truly separate me from testing my own code in which I developed, eliminating biases on both a conscious and subconscious level. Since I did not have access to any interface or explanation as to why all features and their code had specific requirements and constraints, I validated my development and testing concerns by ensuring that the requirements were satisfied. For example, the snippet below is from the code of my AppointmentService.java class which tests the error handling and the requirements of the appointment description, which cannot exceed 50 characters: 

Throughout this project, no value was more important during the testing phases than discipline and the commitment to software quality. It was important to ensure that all JUnit tests passed with 100% testing coverage, that all of the requirements of both the object and service classes were, and that all test classes were accurate and concise with their testing parameters. For example, in the object classes for each feature, every appointment ID, contact ID, and task ID had the same requirements: not null, unique, and not exceed 10 characters in length. Exercising discipline means that each of these requirements is completely satisfied and that no corners are cut, for example, only 9 characters or more for the feature ID variables. By taking the time to validate and confirm 100% accuracy in the testing phases of a program’s code, one can proactively avoid the possibility of technical debt, such as a redesign, overly frequent coding fixes, or unplanned downtime.

References:

Morgan, P., Samaroo, A., Thompson, G., Williams, P., &amp; Hambling, B. (2019). *Software testing: An Istqb-bcs Certified Tester*. BCS, The Charted Institute for IT.