Southern New Hampshire University

Project Two

Benjamin Smith

Dr. Karl Lewis

CS-320-T3330 Software Test Automation & QA

February 19, 2023

I recently developed a mobile application that provided a contact, task, and appointment service. Each of these services had different software requirements and required comprehensive Junit tests. The Junit tests gave confidence that the delivered product was working correctly based on the software requirements and was robust enough to handle error conditions.

The contact service software requirements included the ability to add, delete and update contact objects stored in an in-memory data structure. A contact object required five attributes that would store the contact ID, first name, last name, phone number, and address. None of the attributes were allowed to be null. The contact ID, first name, and last name could not be longer than ten characters. The phone number had to be exactly ten characters and the address could not be longer than thirty characters. All the attributes needed to be updatable except the contact ID. Based on these requirements, I ensured to that the contact object had getters for every attribute and setters for every attribute except the contact ID. This allowed every attribute to be read and only the attributes allowed to be updated could be changed. I included a contact object constructor that would require all the attributes to be populated. The contact service provided the ability to retrieve a singleton ensuring that any interaction with the contact service would be operating on the same in-memory data structure. I chose to represent the in-memory data structure as a map of contact objects with the contact object contact ID as the key. This allowed for a fast look up of existing contact objects. The addContact method would check that a contact object did not exist before adding it. The deleteContact method would check that a contact object did exist before deleting it. The contact service also provides several methods for updating the attributes of contact objects currently being stored in the in-memory data structure. These methods would require the user to provide the contact ID of the contact object in question and the updated value of the attribute in question.

The task service software requirements included the ability to add, delete and update task objects stored in an in-memory data structure. A task object required three attributes that would store the task ID, name, and description. None of the attributes were allowed to be null. The task ID could not be longer than ten characters. The name could not be longer than twenty characters and the description could not be longer than fifty characters. All the attributes needed to be updateable except the task ID. Based on these requirements, I ensured that the task object had getters for every attribute and setters for every attribute except the task ID. This allowed every attribute to be read and only the attributes allowed to be updated could be changed. I included a task object constructor that would require all the attributes to be populated. The task service provided the ability to retrieve a singleton ensuring that any interaction with the task service would be operating on the same in-memory data structure. I chose to represent the in-memory data structure as a map of task objects with the task object task ID as the key. This allowed for a fast look up of existing task objects. The addTask method would check that a task object did not exist before adding it. The deleteTask method would check that a task object did exist before deleting it. The task service also provides several methods for updating the attributes of task objects currently being stored in the in-memory data structure. These methods would require the user to provide the task ID of the task object in question and the updated value of the attribute in question.

The appointment service software requirements included the ability to add, delete and update appointment objects stored in an in-memory data structure. An appointment object required three attributes that would store the appointment ID, appointment date, and description. None of the attributes were allowed to be null. The appointment ID could not be longer than ten characters. The appointment date could not be a date in the past and the description could not be longer than fifty characters. All the attributes needed to be updateable except the appointment ID. Based on these requirements, I ensured that the appointment object had getters for every attribute and setters for every attribute except the appointment ID. This allowed every attribute to be read and only the attributes allowed to be updated could be changed. I included an appointment object constructor that would require all the attributes to be populated. The appointment service provided the ability to retrieve a singleton ensuring that any interaction with the appointment service would be operating on the same in-memory data structure. I chose to represent the in-memory data structure as a map of appointment objects with the appointment object appointment ID as the key. This allowed for a fast look up of existing appointment objects. The addAppointment method would check that an appointment object did not exist before adding it. The deleteAppointment method would check that an appointment object did exist before deleting it. The appointment service also provides several methods for updating the attributes of appointment objects currently being stored in the in-memory data structure. These methods would require the user to provide the appointment ID of the appointment object in question and the updated value of the attribute in question.

The contact, task, and appointment services and their related objects were thoroughly tested using Junit tests. The contact, task, and appointment objects had corresponding test classes that tested the constructors, getters, and setters. The contact, task and appointment services had corresponding test classes that tested the ability to add, delete, and update attributes of the related service objects. These tests would check for both valid and invalid inputs. The invalid inputs for string related attributes included the nulls, minimum length not met, and maximum length not met. For the appointment date attribute in the appointment object, the invalid inputs included nulls, and a date that is in the past. The unit tests for these six classes were deemed effective after running a coverage report. The coverage report indicated that 84.2% of the instructions in the code based were covered. “Higher code coverage increases your chances of finding bugs. And while code coverage doesn’t guarantee perfection, you’ll be significantly less effective without it. Put simply, code coverage tells you how much of your code your tests are reaching. 80% code coverage means 80% of your code is executed during test runs.” (Konik, 2021).

I ensured that my code was technically sound by including unit tests that tested both valid inputs and invalid inputs. Both the definition of valid inputs and invalid inputs were derived from the software requirements. I ensured that valid inputs were processed successfully.   
Graphical user interface, text, application

Description automatically generated

I ensured that the invalid inputs would be successfully handled and returned false indicating that the request failed.   
Graphical user interface, text, application

Description automatically generated

I ensured that my code was efficient by extracting common functionality and using static variables that could be reused. While testing the services, I would use tags that would run common code before each test and ensure a clean testing environment after each test.

Graphical user interface, text, application

Description automatically generated

The use of static variables was used to eliminate typos and ensure the validity of both valid and invalid inputs. This practice also ensures that future changes to valid and invalid inputs will require very little change to the tests.

Text

Description automatically generated

The software testing techniques that I employed for the project included black-box and white-box testing. “Black-box testing (also known as functional or behavioral testing) is based on requirements with no knowledge of the internal program structure or data” (Garcia, 2017). This testing technique was used to develop test cases based on the requirements rather than on the actual code. Black-box testing ensures that tests are aligned with the specifications of the system. This testing technique gives the development team confidence that every requirement outlined by the client is accounted for. “White-box testing (also known as structural testing) is based on knowledge of the internal logic of an application's code” (Garcia, 2017). This testing technique was used to ensure the code developed had adequate code coverage. White-box testing was also used to create tests that would inject faults and ensure they were handled correctly. This testing technique is valuable because it validates the code base is robust and can handle situations not explicitly outlined in the requirements.

The other software testing techniques that I did not use for this project were non-functional testing techniques. These techniques included performance testing, stress testing, and volume testing. “Performance testing may measure response time with a single user exercising the system or with multiple users exercising the system” (Garcia, 2017). “Load testing is focused on increasing the load on the system to some stated or implied maximum load, to verify the system can handle the defined system boundaries” (Garcia, 2017). “Stress testing exercises beyond normal operational capacity to the extent that the system fails, identifying actual boundaries at which the system breaks” (Garcia, 2017). These testing techniques require knowledge of how and where the software will be deployed. Without this information a duplication of work will occur once this information is obtained.

Black-box testing is important because it allows tests to be developed based solely on specifications. This can give confidence when code is completed because a development team will know that their code base meets the expected specifications. White-box testing is important because it tests the actual code base. This technique is perfect for ensuring that code coverage is adequate, and the code is robust enough to handle fault input. Non-functional testing techniques ensure that the software limitations are known and documented.

The mindset I adopted while working on the project was to create tests that not only found bugs but prevented them. I employed caution by ensuring that all software requirements were present in the unit tests. I also looked at testing edge cases with invalid input. It was important to appreciate the complexity and interrelationships of the code to ensure that the tests would not return false positive or negatives. In the service test class, I would use a teardown method after every test. This would ensure that the test environment for each test would be clean and not be polluted by previous tests.

I attempted to limit my bias in reviewing the code base by pretending I had not written the code being tested. This allowed me to write tests that were not tailored to the code base. I can imagine that bias would be a concern while testing your own code. This is because it can be tempting to view completed code as perfect. Instead, unit tests are another opportunity to test your code and refactor the code when needed.

Being disciplined in my commitment to quality as a software engineer professional is very important. Cutting corners when writing or testing code leads to an unstable code base. Every new feature becomes harder and harder to implement when a code base is taped together. Delivering a code base with inadequate test coverage means that a development team cannot say with confidence that the product being delivered is of the highest quality. Being diligent in applying industry best practices will reduce technical debt and allow for less speed bumps during the development process.

References

Garcia, B. (2017). *Mastering software testing with junit 5*. Packt Publishing.

Konik, J. (2021, August 24). *Who cares about code coverage and why?* Codecov. Retrieved February 26, 2023, from https://about.codecov.io/blog/who-cares-about-code-coverage-and-why/#:~:text=Put%20simply%2C%20code%20coverage%20tells,emphasis%20should%20be%20on%20improvement.