CS 320 – Project Two

Clay Dodson

4/15/2022

Concerning my testing approach for project one, I had a similar approach for each of the three features in the project (contact, task, and appointment). JUnit test cases were developed to test the functions in each class. Each JUnit test had examples/cases that it would run through the class functions to determine if the outcome was as expected. For the contact class, I had to test adding new contacts to the database. The main aspects being tested were the getters and setters for the ID, first name, last name, phone, and address. The test would confirm if they fit the requirements of the system (e.g. could not be null and had to be within a certain character limit). The contact service test was designed to test the public and private methods for adding, updating, and deleting contact information.

Similarly, the task class and task service class used test cases to test their functionality. The task class test was designed to the getters and setters for task ID, task name, and task description and whether or not they were null and fit the requirements of the project. Just as in the contact class, illegal argument exceptions will be thrown if the requirements are not met. The task service used a hash map, so it was tested by checking if the task existed in the hash map and if the task name/description was properly updated or if the task was deleted.

Furthermore, the appointment class and appointment service class were tested in a similar fashion. The appointment class was tested to see if inputs for appointments, dates, and descriptions would throw exceptions if they were null or had too many characters. The appointment service class was tested for successful public and private methods for adding/deleting appointments, searching for valid appointments with a sorting algorithm, and getting a list of appointments. The JUnit tests then show the amount of coverage (i.e. percentage of cases that pass the tests) as an indication of the classes and functions performance.

I took every measure to make sure my classes and tests met the provided requirements. Every component of each class was tested to see if it was null. For the contact class, tests were run to determine if the first and last name and ID were longer than 10 characters, if the phone number was exactly 10 characters, and if the address was longer than 30 characters. If any of these exceptions were met, then the system would throw and error and have the user try again. For the contact service class, tests were run to ensure that the hash map would return the needed information accurately and if it would properly add/delete/update contacts. The task class was tested to see if the strings taskID, name, and description were null or longer than 10 characters. Similar to the contact service class, the task service class uses a hash map to store the taskID, name, and description. The hash map is tested to determine if it properly adds/updates/deletes task data and if it returns the correct task data. The appointment class is a bit different in that I had to use a unique test for the date which proved to be difficult. The appointment class is tested to see if the appointment ID is null or longer than 10 characters, the appointment description is null or longer than 50 characters, and if the appointment date is null or in the past. The appointment service class uses an array to store the appointment data. Tests were designed to see if appointments were correctly added/deleted and if the search algorithm located the correct appointment information within the array.

Overall, the JUnit tests were rather effective. The coverage for the contact class was 68.6% and the coverage for the contact service was only 61.5% which was the lowest of all the test results. In looking at the JUnit results in the code, it’s not easy to identify why the coverage was so low but it’s likely the hash map is not configured properly. The task class had coverage of 71.8% and the task service class was 66.9%. Finally, the appointment class was 85.8% and appointment service class was 92%. This leads me to believe there were some issues with my hash map since I ended up using an array for the appointment classes and it resulted in much better coverage overall.

To ensure better coverage, I had to make sure my code was technically sounds and efficient. To be technically sound, each class need to have corresponding functions in the service class that, when called, would run a check to make sure the requirements of the first class were met. For example, in the contact service class, updateFirstName would check the contact ID against the requirements of the setFirstName function in the contact class to make sure it’s not null or longer than 10 characters. Also, to be sound and secure, encapsulation was used to protect user data. Private variables for sensitive information were used to protect the data from exposure. Public functions were used so they can be called across classes with the variable data remaining secure. To ensure the code was efficient, simple data structures were used to store user input for each contact, task, and appointment. Using algorithms like hash maps and arrays allows for simple storing and accessing of the data which doesn’t bog down the system.

For the Contact, Contact Service, Task, Task Service, Appointment, and Appointment Service tests, I employed the black box testing techniques of boundary values and equivalence partitioning. Boundary values were used because the JUnit tests were made to test strict requirements for identification numbers, names, dates, and descriptions which have an upper and lower boundary. Each variable (ID, name, description, etc.) has strict requirements for being acceptable by the program and passing the JUnit test. For example, each name and ID cannot be longer than 10 characters. In this example, 10 is the boundary which cannot be exceeded. Also, the variables cannot be null, so 0 is also a boundary which cannot be breached.

Equivalence partitioning is another technique used in the contact, task, and appointment classes. In limiting the ID, name, and description variables to not being null and being less than a certain number of characters, the program requirements set partitions by which the inputs must fall within in order to pass the tests and be accepted by the program. The JUnit tests check to confirm that inputs are within the set partitions. For example, the Contact name cannot be null (first partition) and cannot be greater than 10 characters in length (second partition). In this way, the program requirements set the partitions and the JUnit tests confirm the inputs fall within the given partitions. Other than black box testing techniques, the white box testing technique of statement testing is also used. Since the Contact Service, Task Service, and Appointment Service classes contain if statements, then the JUnit tests are also conducting statement testing to make sure the if statements are executing as expected.

The black box testing techniques not utilized in these modules, were decision table testing and state transition testing. Decision tables were not used because the program does not accept multiple scenarios for each input, just simple I/O that must meet upper and lower boundary requirements. State transition testing is not used because the program requirements do not create a situation where one input would change the entire state of the program. There are no sequences to be tested in this program because it’s only input/output with simple use cases and not necessarily an entire system to be observed and tested in different states of transition.

The white box testing techniques not used in these modules were flow charts, flow graphs, and decision testing. Since we were not testing an overall system that has interdependent classes and dependencies, then flow charts and flow graphs were not needed to test the program. The test classes are testing the class they are directly linked to (e.g. Contact class and ContactTest class) and nothing further. There is not a sequence of events to observe and test either, so flow charts and graphs would be overkill and not reveal a lot of information other than showing the link between the Task class functions/variables and TaskTest class functions/variables for example. Furthermore, decision testing is not used because the program does not use complex “if, then, else” statements or “do, while” statements. The program only uses simple if statements to confirm if certain requirements are met, so decision testing would not yield very revealing results other than confirming that the simple if statements are working properly which would already be confirmed through the boundary value testing and equivalence partitioning.

As far as my mindset while working on this project, I tried to stay focused on making the JUnit tests successful rather than getting bogged down with the Java application code itself. So, initially, I tried to keep the Java code as simple as possible, but it occurred to me by the appointment service module that my JUnit tests were not getting the coverage that was desired. At this point, I focused more on the way the classes interacted with each other (e.g. the appointment class and appointment service class) and the interdependence of the two in terms of the variables and functions. This led me to try using an array to keep the storage of data much more straightforward. For some reason, the hash map wasn’t getting as good of coverage (61%-71%) as the array (85%-92%). It’s possible there could have been collisions with the hash map for some instances of the test.

Of course, there is going to be bias any time you are testing your own creation whether it’s code, a product, or new recipe. Trying to eliminate that bias is a key skill for software testers. In order to be a successful tester, you have to approach the code from the standpoint of the user and client for whom the software is being developed. You may not like a feature of the software, but a user may like it or the client desire that the feature be included in the final product. Either way, you are tasked with developing code/tests for making sure that feature runs as intended. It would be unethical to cut corners and not develop quality tests or run tests to obtain the best coverage simply because you did not agree with how the code was developed or the requirements of the project.

Each developer/tester has a reputation and it’s based on the quality of their work product. This, in turn, has an impact on the reputations of the company they work for and the company who they develop programs for. Concerning my own personal approach, I will always commit to doing what is required to deliver upon the client’s requests and requirements while making sure the user and their data are protected.

**Works Cited:**

Hambling, B., Samaroo, P., Thompson, A., and Williams, G. (2015). *Software Testing - An ISTQB-BCS Certified Tester Foundation Guide* (3rd Edition). BCS The Chartered Institute for IT.