CS320 T3689

Sarah Kerr

Project Two Report

**Summary**

The tests that were created for the three features of the Contact Service Program meet the software requirements in a twofold approach: by testing that class instantiations occur as expected and by testing that invalid input throws an exception error. Taking the Contact class as an example, the requirements allow for the creation of a new contact as long as it is not null or no longer than the length specified in the requirements. The testAddContact() test checks that allowed input creates a new Contact object and there are a series of tests for exception throws which test for both invalid input (too many characters) as well as null input. Likewise, the tests for the ContactService test the addContact(), deleteContact(), and updateContact() methods as well as testing for exception throws where appropriate. These tests align with the requirements because the program needs to create the Contact, Task, and Appointment objects while also checking the validity of the input. The program also needs to be able to modify the features by deleting them and updating them. The tests created assure this occurs but does not test anything that is not outlined in the specifications (such as ensuring the phone number only contains digits, or that the first and last name only contain letters). The use of code coverage helped to ensure that nearly all of the code was going through a test, but it was not the sole standard of testing completeness. For example, additional test statements were created to check for null input as well as incorrect length since this was specifically outlined in the requirements. However, the addition of these lines of test did not change the code coverage percentage.

The test code is technically sound because it all compiles and runs at it should. Some test parameters were deliberately changed while checking the test units to make sure that invalid input caused the test to fail. For example, within tests for Contact.java, some input was added that contained more than 10 characters so it could be observed with certainty that the tests were failing (this was for experimental purposes only and all tests are written to pass in the final program). Another important feature was checking to ensure the successful deletion of objects. For example, in the TaskService.java tests (lines 36 – 62), three separate Task objects were created and then the first and third ones were deleted. This checked not only the successful deletion of more than one object, but also ensured objects didn’t get accidentally deleted along with the ones that were meant to be deleted. During the first iteration of creating these tests, the test for deleting a Task object only created and deleted one instance. This worked fine when only one object existed, but it failed to catch a bug within the deleteTask() method that was caused by no index variable being used. This caused the entire array holding the Task objects to be deleted rather than the Task object at the found index.

One way code was used efficiently was by using the @BeforeEach annotation within the tests to create String variables to hold the parameters. This annotation was used in each test class, and reduced the need to type in long strings of code for each test method. More importantly, it will also make it easier to update the tests if the requirements change (for example, if the customer wants to change the task service to accept a task description of up to 100 characters rather than 50 all one would have to do is update the description variable.).

**Reflection – Testing Techniques**

For this program, specification based/Black Box techniques were primarily used with the customer requirements acting as the blueprint for the tests that needed to be created. Boundary Value Analysis, which tests at the boundaries of a given range, was used to make sure necessary variables were not null (such as IDs, names, dates, and descriptions) and did not exceed the character limit specified in the requirements. Therefore, the tests for this program tested just at the boundaries of those values. If an ID could not be more than 10 characters, the tests were run with variables having exactly 10 characters to test that boundary. This goes hand-in-hand with equivalence partitioning, which is the idea that one does not need to test every valid and invalid input but rather test groups of input. Therefore, additional tests for ID input that was less than 10 characters are not necessary because a test that passes at the boundary will pass when input is given with fewer characters. Testing at the boundary is important because this is where an error in the code could be caught such as using (ID.length() > 10) rather than (ID.length() >= 10) where the second expression would pass but the first would not. This error would only be found by testing at the boundary. For any software application, these two techniques should be used to test that the program is only allowing acceptable input.

Another technique that was used in this program was state transition testing which is used to make sure output is appropriately affected by specific input. This technique was used to test the methods within the Service part of this program. For example, the method addAppointment() was tested by first creating an appointment object and then making sure the output matched the input. Similarly, the deleteAppointment() method was tested by creating some appointment objects and then deleting only a few of them. This ensured that only the specified appointment objects were being deleted and not all of them. Broadly speaking, state transition testing can be used to test any time there is a change in the state of the program – whether by input being added or an internal change happening. It is useful in making sure the program behaves as predicted.

Structure based/White Box testing was also used in the form of Junit unit tests. Being able to see the source code meant being able to create tests with the goal being full test coverage. Statement testing was used run a test through each statement at least once. As an example, tests for the task service were created for each of the service’s methods: addTask(), getTask(), updateTask(), and deleteTask(). While achieving full code coverage does not guarantee a bug-free program, the goal of structure based testing is full code coverage because it means all parts of the program have been accessed and tested.

Other testing techniques that exist but were not used in the Contact Service program were decision table testing and use case testing. Decision table testing is generally used when there are several inputs and the program behaves differently based on what input is given. This testing method can be used anytime the program has a lot of decisions to make to ensure the program responds correctly to any combination of input. Use case testing essentially tests how the system interacts with the user from the perspective of the user. This means testing each step that the user is supposed to go through occurs as predicted. This is useful in making sure the program runs as expected from the perspective of the user and should be utilized when there is a user-interface.

**Reflection – Mindset**

When creating software tests, it is important to adapt the mindset that every program will have bugs and that it is the tester’s job to find those bugs. While the goal of a software developer is to create bug-free code, the goal of the tester is to reveal bugs within programs. This is why the person doing the final testing of a program should not be the same person who developed it. However, this does not mean that the developer should not test their own programs as doing so helps the developer write better programs and can cut back on time and costs the program incurs in the formal testing process. Caution was specifically taken when creating the Service features of this program to create more than one object so it could be assured that the program was working as expected with multiple objects. This can be seen in the delete methods (deleteAppointment(), deleteTask(), etc) where multiple objects were created and then only some were deleted. Likewise, bias was limited by creating test units for every method and constructor in the program even though so the 3 features were very similar to each other. It was not assumed that just because the Contact.java tests passed that this meant the Task.java and Appointment.java did not have to be tested. However, in order to avoid bias completely, it would be ideal to have the program tested by someone who was not involved in its development.

Overall, being committed to the quality of programs relies heavily on testing and testing early. The sooner a problem can be detected (or better yet, prevented from happening in the first place), the more that time and costs can be saved later. This is why it is important not to cut corners with testing. As mentioned above, assuming a program works correctly and does not need to be tested or not testing thoroughly enough can cause errors to go undetected. This can be expensive and inconvenient at best, and deadly at worst. Software developers can avoid technical debt in practice by not only testing their own programs (and testing early!) but by also having the program tested by someone who was not involved in the development process.