Project Two

7-2: Project Two

Alice Norris

CS-320-T3689 Software Test Automation & QA

Professor Kalinowski

02/21/2022

# Contents

Contents

[Table of Contents 2](#_Toc96360319)

[References 2](#_Toc96360320)

# Summary

## Unit Testing Approach

When planning the unit tests, each test was designed specifically around the requirements for each aspect of the service. Each service (the Contact, Appointment, and Task services) had required functionalities to be implemented, which for each service was a subset of the basic persistent storage operations: create, read, update, and delete (Sulemani, 2021). The creation and deletion functionalities were required across all services, with the ability to update the fields of the task and contact objects required for their respective services. Further, the requirement was stated that each service needed its own in-memory data structure to store instances of the objects for each service. Thus, the tests for each service made sure that created objects were stored in memory by adding them to the data structure and then retrieving the newly added object and ensuring that the input values used in creating the object were the same as their stored value.

Similarly, the specification stated that the fields for each Task, Contact, and Appointment class had a character limit, and could not be null. In the case of the Contact class, the phone number had to be a string of exactly ten characters and could not be null. The tests for each of these fields when set, or upon object creation, ensured that exceptions were thrown if the specific character limit for each field was exceeded, or if the data supplied to the constructor or update method exceeded the limit or was null. Both valid inputs (a string under the character limit or of the specified length, or in the case of Contact phone numbers, a string of ten characters) and nonvalid inputs (null and excessively long or insufficient character) were tested, with the limitations directly gathered from the specification. Testing efficacy was measured using a coverage metric combined with the fact that all stated requirements were tested thoroughly. Since the tests covered all possible requirements, and the tests tested much of the code at 95% coverage, it can be stated with reasonable confidence that the tests thoroughly covered the code, and the code met the stated requirements.

## Writing Unit Tests

As each service and its respective object had similar functionalities described by similar specifications, the testing was likewise similar in each case. Testing for creating instances of Contact, Task, and Appointment objects was accomplished by first setting up valid variables to act as constructor arguments, as can be found in lines 14-18 of ContactTest.java, 14-16 of TaskTest.java, and lines 15-17 of ApptTest.java, respectively (Norris, 2022). The constructors were then tested for valid and nonvalid input.

For Contact objects, valid constructor tests were done at lines 28-35, where Contact object was created with valid arguments to the constructor and the resulting objects’ fields tested for equivalence to the given parameters. Nonvalid inputs were given to the Contact constructor in the functions from lines 92-145 in ContactTest.java, where each parameter was given as violating character limits or null, ensuring that exceptions with appropriate messages were thrown for each (Norris, 2022). Similar tests were done for the constructors for Task, with a valid constructor test at lines 26-31, and nonvalid constructor tests at lines 70-101 in TaskTest.java. Tests for valid and nonvalid Appointment constructors can be found in ApptTest.java, at lines 27-33 and 74-109 respectively (Norris, 2022). Since the setters are used in the constructor for each to validate input, these tests also technically test these functions, but they are tested in the remainder of the class tests regardless.

When testing each service for valid inputs as far as creation and deletion of objects held in the services’ respective memory structures, an instance of the service was created, along with the declaration of variables for instantiation of an object for each service. This can be seen in lines 17-35 of ApptServiceTest.java, 16-37 of ContactServiceTest.java, and 16-36 of TaskServiceTest.java (Norris, 2022). The valid creation operation tests for each class simply created instances of each object (Contact, Task, Appointment), added them to their in-memory data structures, and then retrieved their fields, ensuring they were the same as the parameters given previously. This can be found in lines 47-52 of TaskServiceTest.java, 47-54 of ContactServiceTest.java, and 46-51 of ApptServiceTest.java (Norris, 2022). Nonvalid creation and addition of classes can be found in lines 71-90 of ApptServiceTest.java, 126-175 of ContactServiceTest.java, and 107-129 of TaskServiceTest.java. In each case, excessively long, excessively short, or null parameters were tested, and their exception messages compared to the expected message to make sure that nonvalid input was caught. In the remainder of these code files, the update functionality was tested, using both valid and nonvalid inputs (Norris, 2022). In this way, both the services and their classes were tested for proper operation. As no metrics or requirements for efficiency were given, no efficiency testing was done. However, at the risk of echoing the phrase “it ran on my machine”, in my testing environment all 72 tests can be run in about 100 milliseconds.

# Reflection

Similarly to design of the unit tests, the testing techniques chosen were similar across the services, as well as the techniques for the corresponding classes. Testing techniques generally fall in one of two varieties: specification-based testing and structure-based techniques ((Hambling et al., 2015, p. 78). Primarily, specification-based techniques were used in testing this code, specifically equivalence partitioning, boundary value analysis, and decision table testing. The two structure-based techniques used were simply measurement of decision and code coverage metrics. In equivalence partitioning, partitions of input “types” are determined, and representative values from each partition is chosen and tested, with the assumption that any other value that falls in that partition will act similarly in testing (Hambling et al., 2015, p. 87). This goes together with boundary value analysis and decision table testing. In boundary value analysis, values are chosen that fall on either side of a ‘boundary’, where defects tend to accumulate (Hambling et al., 2015, pp. 89-90) such as the testing done for this code, where values were chosen that were just outside or just inside the character limits. This follows with decision table testing, in which input conditions that can occur are mapped and the combinations tested for proper flow (Hambling et al., 2015, pp. 91). By choosing values on the edge of boundaries, this tested decisions on whether to throw exceptions. Code coverage can be used in conjunction with well-defined tests to show that decisions and tests hit all the code and that all code is used as needed (Hambling et al., 2015, p. 83). These techniques are useful for almost all projects, as decisions and input validation are almost always necessary. Unused techniques include flowchart, control flow, and use-case diagramming, in which diagrams are made to represent the flow of decisions and control across different components of software (Hambling et al., 2015, p. 100). These would be more useful in situations where software is more complex or has a larger development team.

When writing tests, I both used care and eliminated bias by completely changing my mindset. While writing the code, I was careful to make code that worked. However, when testing, I changed my approach to trying to break it. It is important to take this approach, as it is as important to break the code as it is for it to work. If one were to see the code as their “baby”, they may miss defects or errors by employing only tests that are sure to pass. If a programmer were to protect their pride by obscuring or simply ignoring the possibility of error, harm to end user property, or worse, direct harm to end users, could occur.

# References

Hambling, B. M., Samaroo, P., Thompson, A., & Williams, P. G. (2015). *Software Testing - An Istqb-Bcs Certified Tester Foundation Guide* (3rd ed.). BCS The Chartered Institute for IT. Retrieved from https://app.knovel.com/hotlink/toc/id:kpSTAIST01/software-testing-an-istqb/software-testing-an-istqb

Norris, A. (2022). CS320-Project-One [Computer software]. https://github.com/Alice-Norris/CS-320-Project-One

Sulemani, M. (2021, April 7). *CRUD operations explained: Create, read, update, Delete*. Educative. Retrieved February 22, 2022, from https://www.educative.io/blog/crud-operations