**Junit Testing of the Contact and Task Classes**

**CS-320 Module 4**

Damean Murphy-Short

September 19, 2023

This paper serves as a reflection of the testing process used during the creation of the Contact Service and Task Service classes which will be used for an upcoming project. During the process of test creation, the author was pleasantly surprised at the usefulness of such tests, and the reasoning for this will be elaborated below. The paper will also aim to provide objective argument that the tests written were both effective at ensuring delivery of requirements and performance.

By using the integrated J-Unit tools in Eclipse, we observed over 99% coverage of the classes under testing. Whilst high coverage does not guarantee high quality code (Google, et al., 2020), we can at least reason about the observed behaviour of almost every branch and statement of the code and verify that it does satisfy the requirements outlined by the guidelines document. Because there are many guard statements within the classes under test, for instance in many cases there are statements that state if the input is not null, set some attribute to the input, or else set the input to an empty string, and since these guard statements are meant to satisfy some requirement – that the strings not be made null – we can demonstrate through test coverage of these branches that it is not possible to make the strings null, and therefore fulfilling the layed out requirements. The test methodology most closely observed was that of partitioning, values were chosen that were inside and outside of valid requirements to ensure that all cases were tested without exhaustive testing. We can see this when we build values on the edge of valid input like in “build\_boundary\_values” method used @BeforeAll tests and then increment them so that they represent the partition of invalid values.

As can be seen in the comments for each test, the tests are designed to verify one requirement each, lending them to a simple design that effectively balances function and maintainability. Sticking with the example above, all values that shall not be null are tested at once by attempting to assign them as a null string and using an assertAll statement to verify that none of them have become null. In some cases, it made sense to verify multiple requirements in one test, as was the case with creating a unique ID that cannot be changed. Since the environment for the tests is the same: an object without an ID is given an ID, then an attempt to change it is made; the logical conclusion was to merge the tests instead of creating this environment twice. In this way, many of the tests are written in a “black-box” style as a means of requirement verification.

There were also many lines of code dedicated to ensuring technical correctness that come from a more “white box” approach, and some tests were made only to cover these branches where the code is attempting to catch some erroroneous input. For example, a method of retreiving objects from the underlying ArrayList used to store them in memory was devised in the get\_contact/task\_at(index) method. This method must use a try/catch block to ensure that no out-of-bounds value is used as index, which would cause the arrayList to throw an exception that must be caught or else terminate the program. In an attempt to reach greater test coverage, a test was devised to create an environment where this branch would be executed, and we were able to verify that the program does as expected in this case and returns a default constructed object in place of throwing an exception.

Whilst the J-Unit tests all executed in under a second, the author does not believe this to be a true test of the efficiency of the code. Indeed, a static analysis of the code serves here a better test, since it would be difficult to produce a load large enough solely in memory that would reveal weaknesses in code performance. One of the more “time expensive” operations that are being performed is the retreival of objects in the ArrayList by their string ID. We guarantee a linear performance in this task by using an iterator pattern that queries each object in order until it has found a match. We also use the ArrayList data structure supplied by common Java libraries instead of a “homemade” solution due to the assumption that the ArrayList will be optimized well for our particular purposes of storing similar objects in memory and removing them from arbitrary indexes. All other methods in the classes under test are mostly only a few lines each, and don’t use any expensive searching, sorting, or memory allocation, but rather assignment to stack variables using other variables on the stack. As such, their performance should be less scrutinized in comparison to the former.

In conclusion, the author is confident that the tests written provide assurance that the classes written are efficient, meet requirements, and are not prone to undefined behaviour due to invalid input. In this way, we consider the testing succesful, and plan to do more testing in this manner in the future.

**References.**

Google, Bender, A., Arguelles, C., & Ivankovic, M. (2020, August 7). *Code coverage best practices*. Google Testing Blog. https://testing.googleblog.com/2020/08/code-coverage-best-practices.html