**CS 320 Project Two:**

**Techniques and Effectiveness of Testing for Project One**

Damean Murphy-Short

Southern New Hampshire University

This paper aims to serve as a reflection of the testing techniques used to evaluate the Contact, Task, and Appointment classes that compose the mobile app being developed for the “Grand Strand Systems” client. In this report, we will explore which techniques were applied and why they were chosen, whether the tests verify that the software produced adheres to the requirements laid out, the effectiveness of the tests conducted, the extent to which the code produced was covered in testing, and an introspection on the metholodology of the testing. To this end, we hope to uncover the successes and failures of the testing carried out so as to have an improved approach to future testing.

To begin, let us verify that the testing does what it is meant to do at a bare minimum level and analyze whether the requirements of the project had been met. For the most part, despite the tests being designed after the systems under test had been written, a ‘black box’ approach is taken to the input and output of the testing methods. This is done intentionally because almost every test is written with a specific requirement or requirements in mind. For a concrete example, the test\_invalid\_mutations\_null() method, present in ContactTest, TaskTest, and AppointmentTest, simply uses the public API of the appropriate class to try to assign a null value to a certain attribute, and asserts that this always throws an exception. By making each test “responsible” for a requirement in this way, and by creating simple pass/fail assessments of whether the requirement is met, we can more confidently assume that when the tests are passed, the requirements are indeed upheld.

There are some tests that do not explicitly test a requirement, but only seek to prove or disprove the robustness of the architecture chosen to make the requirements possible. For example, the “Service” classes require some data structure to store objects in memory. An ArrayList was chosen for this task, and, knowing this, white box testing techniques were utilized to ensure that the system does not behave erratically or in undefined ways when out of bounds indexes are requested or invalid searches made. In this specific example, the test\_update\_empty\_contacts/tasks/appointments() method is used, knowing that it is possible to call the get\_contact/task/appointment\_at() method using any integer value, to request a value that is outside of the ArrayList bounds. The test asserts that the method returns some defined behaviour of returning an “empty” attribute object rather than some garbage(in the sense that they are random or part of memory that should not be read) values or an unhandled exception that may crash the system.

With the combination of both of these types of tests, a rather high testing coverage was produced, ranging from 87% coverage on the low end to some classes having 100% branch and statement coverage. While the fact that these statements were executed in testing does not by itself mean that the code being run was of great quality, it does assure the tester that a defined behaviour can be demonstrated for each branch and action possible, for up to at least 87% of such branches. This does not eliminate the possibility of defects existing, but does give us more certainty that defects would have been uncovered in combination with the other testing techniques employed, which will be returned to later in the report.

In correlation with our testing results, we have some guarantees about our code that we can assert: null values are not permitted, out of bounds indexes do not crash the program or return garbage values, and values do not exceed their maximum sizes. This speaks to the technical rigidity of the overall program. It can also be said that using well-defined data structures that are built into the JVM like ArrayList and Iterator avoid many of the issues that could arise when creating one’s own data structures, and provide a certain robustness that ensures that at the very least, an exception will be thrown in the face of some unexpected occurrence. For example, take the find\_ID\_index() method from any of the “Service” classes. It uses an iterator to search linearly each element of the underlying private ArrayList for an Object with a matching ID and returns its index or a defined constant NO\_INDEX if it is not contained in the structure. We know the efficiency of this search is linear(and with the data not being necesarrily sorted in the ArrayList, there is not a much better time complexity that can be achieved), and we know that it has defined behaviour for reaching the end of the list, reaching the desired element, and not finding the desired element. Similarly, because we use ArrayList for the underlying data structure, we have a relatively well known time complexity for adding, removing, and querying elements that is somewhere between constant time and linear time depending on how often the ArrayList is grown to accommodate more elements (“ArrayList..”, 2023).

With the established assumption that most statements in the code were tested and that the code uses sound practices to achieve its goals, let us examine the specific techniques with which the tests were developed to ensure this. The most important of these techniques were the input value techniques known as boundary value testing and equivalence partitioning. We see in most of the test classes a method run before all tests called “build\_boundary\_values()” that created variables at the exact length of the maximum accepted for a specific attribute of the class under test. This serves two purposes: create a value within the “valid” partition to test and to create a value that is just inside this boundary of valid values. Then, in the “test\_invalid\_mutations\_length()” class, we change by 1 the size of the variable, placing it in the “invalid” partition just outside of the valid boundaries. In this way, we effectively test for off-by-one errors and have a relatively strong assurance that any valid values will be accepted and all invalid values will be rejected without having to perform exhaustive testing, which would be impossible in the case of the strings at test. These types of testing would be especially important were we using a language with dynamic manual memory allocation, as off-by-one errors can produce difficult-to-debug scenarios in which the program behaves as expected under most cases, but not when these boundary values are encountered.

Some tests that were not performed as part of this project include security testing and performance testing. In a real product that will reach deployment, one cannot skip security testing. In our case, all memory is volatile and contained to an ArrayList, but in a project that uses databases and non-volatile memory, the lack of testing for things like SQL injection in input and attempts to access different areas of memory would be unacceptable. Likewise, a large database may find an unsorted ArrayList with linear searches too slow for the scale of objects that need to be stored and opt to design tests aimed at ensuring that searching, indexing, adding, and removing objects all take place in an acceptable amount of time. For example, a set of objects sorted by the value of their ID’s would be faster to search through using a binary search or some other method, and testing would be required to demonstrate that it is up to scale. It also goes without saying that the classes in question were not integrated into a single system, and so whole system testing and integration testing were mostly off the table.

Finally, a reflection on the testing methodology: the sole author of the classes under test was also the designer of the tests themselves. This, by itself, introduces bias in how the tests were designed, and is in large part responsible for the decision to make the tests “black box” in style to reduce this bias as much as possible. The mutator methods such as set\_ID() often have guard statements to ensure invalid input is not expected like “if (id == null) throw exception;” which will be the same invalid input that is tested for in the test classes. Perhaps a different tester would have imagined more types of invalid inputs. This does give the advantage, however, that the tester knows which calls to the Service classes will proceed to call known methods in the object classes. For example, the “test\_update\_contact/task/appointment()” methods call the update\_firstName() method for the service class, which in turn calls set\_firstName() method of the Contact class. Knowing that the latter method has already been tested allows the tester to be sure that the former method gives a definitive answer to where the error lies if the assertions in the test fail. Also to eliminate bias, it was important of the tester to not assume that the code they had written was correct only because it contained what they consider “good practices” that they themself had written. For example, despite the assumption that out of bound values would return a “default” state of an object when requesting an index that is not contained in the ArrayList, a test was still written to ensure this behaviour, as cited above for the “test\_update\_empty\_object()” method, to avoid ‘cutting corners’ in testing and tying up any loose ends that may cause problems.

In conclusion of this report, we recommend that boundary value testing continue to play a role in future tests, that security testing be more heavily emphasized in future requirements, and that different testers are brought in to eliminate the bias of the software developers, despite their efforts to eliminate such bias. In this case, we are fairly confident in the quality and adherence of the code to stated requirements, and believe that the relationship of tests to specific requirements was executed well.

References.

*ArrayList (Java Platform SE 8 )*. (2023, June 14). https://docs.oracle.com/javase/8/docs/api/java/util/ArrayList.html