Harrison Bergeron

CS 320

Summary and Reflections Report

June 27 2021

The JUnit tests I created aligned with the software design because the fields were tested based on initial requirements like ensuring that the ID, name, and description fields were not null or exceeding the number of characters specified by the design. For example, if the task ID was null when constructing a Task object, an IllegalArgumentException is thrown, and the TaskTest JUnit test covers this case as well as cases for the other arguments. The tests I wrote for the Task Service project were more complete, whereas the Contact and ContactService classes had poor coverage because not all possible branches were achieved by the tests. When running either the TaskTest or TaskServiceTest for coverage, the Task and TaskService classes show one hundred percent coverage, and all possible branches were tested for creating tasks as well as adding, deleting, and updating them. In writing tests for the ContactService class, one way it achieves technical soundness is by ensuring that the proper Contact objects are deleted by the deleteContact method. Two contacts with different ID’s (“123” and “456”) are added to a ContactService object, and then one is deleted followed by a statement to test that one is gone but the other still exists: “assertTrue(!cs.hasContact(contact1.getId()) && cs.hasContact(contact2.getId())).” The TaskService class has improved tests for updating Task records because it not only ensures that a Task object’s fields are updated, but that an exception would be thrown when an invalid ID is used: “assertThrows(IllegalArgumentException.class, () -> { ts.updateName("999", "Task 1"); }).” To make the code more efficient when testing, only key test cases were written. There are many possible permutations of Task and Contact objects that can be constructed, and they can be added and deleted from their respective service classes in many different configurations. Instead of attempting to test every possible combination of data, only key test cases were written to test against string fields being longer than the limit or being null. For example, the task ID field is tested by creating these two objects and testing for IllegalArgumentExceptions: “new Task(null, "Task 1", "This is a task")” and “new Task("01234567891", "Task 1", "This is a task").” In only two lines, the basic requirements for the taskID string field are tested without covering every possibility, which would be too slow.

All three of the milestones were organized such that there was a class that contained information associated with a unique ID, and a service class that managed a list of instances of this class. The former class type was always tested to ensure that data sent to constructors and setters was consistent with data returned by getters, and that the arguments were not null and did not exceed length limits. The service class type was always tested to ensure that objects added had a unique ID, could be verified to exist after being added, and could be verified to not exist after being deleted. All of these testing methods fall under black box testing, because the underlying implementation of the methods is not what was being tested, but simply the output of the methods considering the given input. For example, in adding Appointment objects to the AppointmentService class, the test class did not care what data structure was used to store the appointment object nor what methods were used to insert it into the data structure. The test class was only concerned that once an Appointment object with a given ID was added to the service object, the same ID could be verified to exist within the database later. Other testing techniques that were not used include white box testing and experience-based techniques. White box testing involves testing the structure of the code or the way that different classes interface with each other structurally. Experienced-based testing involves having a human test functionality for potential errors based on what is common for that type of software and past experience with it. White box testing could be useful for larger software projects where integration is being tested, since multiple software units need to be tested for their ability to properly interface with each other. Experience-based testing would be useful for higher level testing of software functionality in cases where a software system is being created that closely resembles many other projects created before it. An appointment booking system is an example of such a system where testers have likely already had extensive experience with testing and could use prior knowledge to create specific error-finding test cases.

As a software tester, it was important to exercise caution at every step of the way. Developing JUnit tests for each of the classes was not straightforward and required an understanding of the program structure. Some aspects of the tests were easy to address, like the testing of variable values after being assigned in constructors or setter methods. On the other hand, there were certain tests that required knowing how the classes operated together to create effective test cases. For example, testing whether Contact objects were correctly added to the ContactService database involved writing code that added and deleted Contact objects from a ContactService object and assessing its success with more complicated conditions aside from simple equality tests. By being careful not to make assumptions about how the code worked under the hood and being cautious not to miss important test cases, effective JUnit tests could be written with one hundred percent coverage. One way I tried to avoid bias in writing tests was to focus on what functionality was being tested, without concerning myself with the efficiency or implementation of the program. Based on requirements that were already specified, I wrote tests for functionality that would indicate that the software worked as intended without considering program structure. For example, once the Appointment and AppointmentService classes were written, I didn’t bother analyzing them any further for proper structure or good design, but only focused on testing the available methods with black box testing techniques. When testing your own code, it is easy to let bias affect your ability to accurately write test cases that cover all bases. It’s easy to forget all of the requirements that need to be tested and only be focused on testing cases that you planned for in your own program design. It is important not to take too many shortcuts when writing code, and to put thought into program design to save time in the future. By focusing on quality, a product can be developed and delivered based on requirements definitions and customer expectations that meets the needs of the user. By trying to save time and cut corners, the resulting product will be flawed and the program will eventually need to be fixed, which can cost a lot of money the further down the production line that a product is. Spending too much time planning can be a negative symptom of the waterfall development model, although not enough planning can ultimately waste time and cost money. It is important to work with other developers and testers on the team to make sure that the requirements of the software product are understood and being implemented correctly.