Derek Nill

Professor Moscon

CS-320 Software Test Automation

February 18, 2021

I recently finished a project for the back-end services of a mobile application. Included in this project were the following classes: Contact, ContactService, Task, TaskService, Appointment, and AppointmentService. Junit 5 was integrated into the project, which spurred the creation of test classes for each of the mentioned classes.

My software testing approach aligned strictly with the software requirements that were given. Test cases were written for every requirement. Most of these requirements involved testing that the code would not allow variables to be longer than the allotted maximum length, nor will it allow them to be null. For example, in the TaskTest class, I wrote the following code, which makes sure that an exception will be thrown if a task is created with an ID that is too long.

*@Test*

public void testTaskIDTooLong() {

*assertThrows*(IllegalArgumentException.class, () -> {

new Task("12345678910", "Write unit tests", "The action of coding unit tests");

});

}

Overall, the quality of the tests for the Contact Service, Task Service, and Appointment Service are very high. I am able to check this by opening the test classes for each of these classes and choosing “coverage as”. This results in a coverage of 100% for all of them. This means that every executable part of the code from each of these classes has been touched by at least one of the tests in the test class.

I ensured that my code was technically sound in a few ways. I made sure that the tests I wrote checked for more than just null inputs or inputs that were too long. For example, in line 31 of the Task Service Test class, I wrote a test method called testDeleteDesk. This method creates a new Task object by using the addTask method of TaskService. It then checks that the newly added task actually exists in the task map by using the getTask method. It uses the deleteTask method to delete the Task, and then uses the getTask method again, this time to check that the Task no longer exists. Here it is shown.

*@Test*

public void testDeleteTask() {

// Create a new Task, then check that the new Task object exists in the Task list map.

TaskService taskService = new TaskService();

taskService.addTask("12345", "Write unit tests", "The action of coding unit tests");

*assertTrue*(taskService.getTask("12345") != null);

// use the deleteTask method to then remove the Task from the map, and then check that it is in fact gone.

taskService.deleteTask("12345");

*assertTrue*(taskService.getTask("12345") == null);

}

The most important strategy that I used in these projects to make sure that my code was efficient was using the proper data structure. The Contact Service, Task Service, and Appointment Service required an in-memory data structure to hold the Contact, Task, and Appointment objects, respectively. At first, I thought that a List would make the most sense for this. When implementing the functionality of the class, I realized that I would often have to find Tasks or Contacts by their ID attribute. If I kept a list, this would mean iterating through the List every time that I needed to find one of these objects. It was then that I realized that a Map using IDs as keys and Objects (Tasks or Contacts) as values would make the most sense for efficiencies sake. This can be seen on line 10 of the Task Service class, which is as follows.

private Map<String, Task> taskMap = new HashMap<String, Task>();

Boundary value analysis is a type of black box testing. The theory behind it is that many errors occur on the boundaries of acceptable values. For example, for each of the Contact, Task, and Appointment classes there were maximum values for setting the string inputs as attributes. The code was written to throw an IllegalArgumentException if these inputs exceeded the maximum value. I wrote Junit tests that attempted to use an input that was only one character larger than the accepted size. If an exception was thrown, then the test passed. A practical use of boundary value analysis is age testing. For example, if a website must ask for a user’s birth date to make sure they are at least 21 years of age, boundary value analysis could be used to see what happens if a birthdate is entered that is only one day younger than 21 years of age.

Decision testing is a type of white box testing and it involves testing the logical decisions made within an application. I used this test to verify that many of the if statements were functioning properly. For example, the Contact, Task, and Appointment classes all contained an ID attribute that was meant to be unique. The ContactService, TaskService, and AppointmentService classes contained methods to add Contacts, Tasks, and Appointments. In these methods, a check was made to make sure that no objects were added with a duplicate ID. I then wrote Junit tests that attempted to use an ID that was already in use. When these tests were run, an exception should have been thrown. A general example of decision testing being used practically is for any business logic in an application. Most applications being build for businesses will specify decisions that the application must make based on inputs. Decision testing would be useful in making sure that these applications are running correctly.

I did not use experience-based testing for any of these classes. One type of experience-based testing is error guessing. Error guessing is a way to target testing likely weaknesses of an application. It is usually best when performed by an experienced tester since a large part of its effectiveness will come from the testers intuition in finding likely weaknesses.

When working on this project, I employed caution when writing the tests. It can be easy for a tester to get carried away. I wrote tests that cover all of the requirements listed in documentation but did not go past that. The ContactTest class is a good example. In this class, I tested that a Contact’s id is not null, and is also not too long. These were the required parameters given. However, I could have spent my time writing many additional tests. For example, I could have checked that the contact id does not contain non-numeric symbols. I also could have checked that the system still functions if the contact id argument that is passed in is extremely too long, such as thousands of characters. Employing caution is important because it saves time as well as prevents false positives.

Bias can become a serious issue when testing software. This is especially true when the person who wrote the code is the same person writing the tests for the code. I tried to limit bias by carefully going over the requirements documentation and making sure that every requirement was tested for. I tried to avoid confirmation bias, which happens when the tester favors selected test cases that which the outcome is likely to meet expectations.

Discipline is an important trait for a software engineer. Writing high quality and well-tested code definitely takes additional time. It can be tempting to cut corners as long as the software is functioning, however this can be dangerous. First, high quality code is extensible and readable, which helps with an applications maintainability as well as future upgrades. Second, well-tested code, while taking more time initially, can save a lot of time and money down the road. Errors caught early on are easily fixed. When they are discovered as defects in a released piece of software, it can be extremely costly to fix.