**Brandon Hobbs**

**CS-320**

**August 10, 2022**

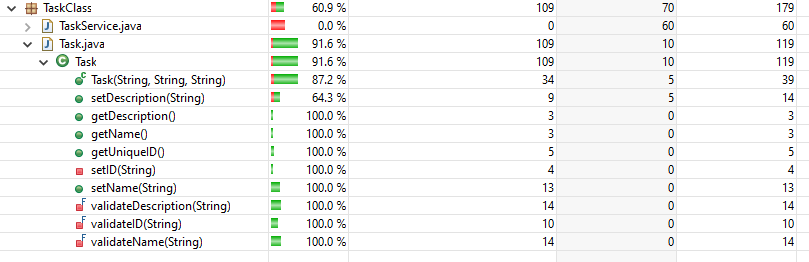
**Project 2**

During the creation of the model and service classes and their subsequent testing via JUnit I adopted two different personas. One persona being that of a software programmer and the other a malicious schemer determined to prove the programmer had no idea what they were doing.

I used Unit Tests to try and test the boundaries of the specification. For example, the specification of the Task says that the full name cannot be null, greater than 20 characters, and is mandatory. To test this specification, no less than 4 branches are needed.  
  
*if (fullName == null || fullName.length() > 20 || fullName.equals("")) {return false; }*  
  
I tested a valid version of the name, a null string, a very long string, and an empty string. Two samples are below:  
  
*Assertions.assertThrows(IllegalArgumentException.class, () -> {tempTask.setName("");*

*Assertions.assertThrows(IllegalArgumentException.class, () -> { tempTask.setName("This name is incredibly long and why is this a name, anyway");*

I did not try for 100% coverage of the Task and Contact classes. The constructors had 3+ parameters to set and I didn’t think it was worthwhile to test good and bad constructors for each of these parameters. Instead, I targeted 100% of the getters and setters and only a good constructor and a single invalid constructor.



To ensure the software was technically sound**,** I tried to test “to the spirt of the code”. For example, in *ContactService* there was to be a method that deleted an ID. This could be tested in a few ways: 1) add three objects to the collection, delete one, and then confirm the collection is now size of two; 2) look through the entire collection for what you just deleted to confirm it is gone.  
  
The issue with only doing one of these is it leaves room for aberrant behavior. If only the collection size is checked then the wrong item could be deleted and the test passes. If the item is searched for as the only technique the test could pass if all the items were deleted accidently. I tested for both:

*@DisplayName("Test deleteContact")*

*@Test* ***void*** *testDeleteContact() {*

*String firstName = "Greg";*

*String lastName = "Hoffman";*

*String phoneNumber = "1234567891";*

*String address = "123 Main Street";*

***boolean*** *testBool =* ***false****;*

*ContactService test =* ***new*** *ContactService();*

*assertTrue(ContactService.contactList.isEmpty());*

*test.addContact(firstName, lastName, phoneNumber, address);//object ID 0*

*test.addContact(firstName, lastName, phoneNumber, address);//object ID 1*

*test.addContact(firstName, lastName, phoneNumber, address);//object ID 2*

*assertEquals(3,ContactService.contactList.size());*

*test.deleteContact("1");*

*assertEquals(2,ContactService.contactList.size());* ***//test one***

*//loop through to look for ID*

***//Test 2***

***for****(****int*** *i = 0; i < ContactService.contactList.size(); i++) {*

***if****(ContactService.contactList.get(i).getContactID() == 1) {*

*testBool =* ***true****;*

*}*

*}*

*assertFalse(testBool);*

*}*

As mentioned above, it’s not enough to just check the size of a collection after a delete, checking that it is actually deleted is important. One step further could be to ensure the remaining objects are still intact and as expected (this was not tested for in my code).

I did, however, test to make sure objects were instantiated only when requested:

*ContactService test =* ***new*** *ContactService();*

*assertTrue(ContactService.contactList.isEmpty());*

I also checked that after a failed string update the string was still as before the attempt:

*tempTask.updateTasks("1", fullName, "New description"); //bad ID*

*assertNotEquals("New description", TaskService.tasks.get(id).getDescription());*

*assertEquals(fullName, TaskService.tasks.get(id).getName()); //original string*

For the 3 modules needed in this app, I employed unit testing and static testing - both of which are a form of whitebox testing. Static testing involves studying the code and comparing it to the specifications in an attempt to identify bugs. I used this technique to find and fix the offending code the code after one of my JUnit tests failed.  
  
As I was writing the class methods, I was attempting to design the logic to match the specification document. Then, during the unit test creation I re-read the specification and tried to find ways to prove that I made an error during coding. A few times one of the JUnit tests failed and I had to inspect the static code to figure out why. In one such case, I had used the wrong attribute within the logic branch – so of course the test failed.  
  
I also figured out that merely making sure an exception is thrown when expected is not enough. Catching the correct and expected exception is also important. For example, in another case I was asserting that a constructor would fail due to bad input. An error **was** thrown but not the one I meant to be thrown. So, in subsequent code I started evaluating the exception type to ensure it was the exception I expected, and my test is not creating a false-positive.

The service classes have very little system testing and no integration tests. The specifications created a model class and service class that drives and interacts with the model class. In theory, you can test the model class through the service class and treat this as a system. I did a bit of this in later developed code – this missing testing should be treated as technical debt.

Integration testing would test the entire application, bringing all the systems together for testing. In this case, the system and integration tests would be the same. I could, in theory, validate that the *TaskService*, *AppointmentService*, and any other classes all work together well at their integration layer – I have not.  
I also did no automated testing – all my JUnit tests were manually run. In theory, I could have had a service that ran the JUnit tests after each build-event in Eclipse. Automated tests are important for large applications and should be built into a continuous integration pipeline – especially if continuous delivery is being practiced. Automated tests can be at the build stage, pre-deployment, or even post deployment.

There was also no security scanning performed. Libraries and components need to be scanned for security vulnerabilities. Security Testing is always important, especially where data is concerned. Right now, there are no databases and no user input into the system so protecting against improperly formatted input, e.g., SQL injection or buffer overruns, is not needed.

Unit testing is always useful and can help to ensure that small issues do not become huge problems. Unit tests can also help to harden the code against missed logic branches, e.g., what is the value is null, and missed specifications, e.g., maximum string length is supposed to be 50 characters. System and Integration tests would become useful as the code base enlarges and there are more interconnected components the more important these flavors of testing become.

My mindset in each phase was slightly different as I mentioned during my opening. During the coding phase I tried to interpret the true intention of the requirement and not just what was written. For example, the Appointment class had this requirement:

*The appointment object shall have a required description String field that cannot be longer than 50 characters. The description field shall not be null*.

Simply read, this means there should be a logic test for length and if the passed argument is null. But the spirt of the requirement suggests that there MUST be a description. So, it stands to reason that the description should also not be a blank/empty string, thus the validation test becomes:

*private final boolean validateDescription(String description) {*

*if (description == null || description.length() > 50 || description.equals("")) {*

*return false;*

*}*

*return true;}*

Also, when looking at how the Service classes are intended to use the Model classes I determined that using a Hash Map was more efficient than a List. Instead of having to create the code that searches the collection for an index I could simply use the Hash Map method contains. See below:

Use of a List collection in the remove method (have to loop through the entire collection)

***for****(****int*** *i = 0; i < ContactService.contactList.size(); i++) {*

***if****(ContactService.contactList.get(i).getContactID() == intID) {*

*contactList.remove(i);*

*}*

*}*

Use of a Hash Map (no looping needed)

***if****(tasks.containsKey(\_ID)) {*

*tasks.remove(\_ID);*

*}*

The use of the Hash Map also simplified the JUnit code.

The main bias I tried to remove, or at least test for, was that of “I know I wrote the code correctly”. As Maaret Pyhajarvi said, “Tests don’t break your code, they break your illusions about the quality of that code.” (<https://www.quora.com/What-are-your-favorite-software-engineering-quotes>). Instead of just testing that any exception was thrown, I used the JUnit tests to check if the correct exception was thrown. See below:

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

*AppointmentService tempAppt =* ***new*** *AppointmentService();*

*tempAppt.addUniqueAppointment(goodDate, description);*

*});*

*assertEquals("Invalid description", exception.getMessage());*

The avoidance of Technical Debt is a long journey and not a short trip. This theme has to be revisited at regular intervals. Security issues arise every day and exploits can be attempted in mere hours. Moreover, technical debt isn’t always a bug it might be a missed use case. Continuing to measure user satisfaction, i.e., consider tools such as UserVoice, or fixing issues found with static and dynamic scanning tools built into pipelines, SonarQube is quite popular, or even improving code styling and quality, i.e., use Lint Code, take quite a bit of time within commercially sustained software. I typically estimate no-less-than 30% of an engineer’s time for these activities.

Beyond the obvious benefits of the JUnit tests, they can also serve as an indicator of code quality. Often more mature software has higher code coverage which can lead to better sustainability. Low code coverage should be treated as technical debt. Within this app the JUnit tests provided a “second set of eyes” and caught a few cases of improper copy/paste. To get the most benefit from the unit tests it’s imperative that the coder puts aside pride and set out to prove what’s intended actually occurs with no side effects.