Kevin Schroeder

Summary

My primary goal when developing the Contact, ContactService, Task, and TaskService classes, my primary goals was to satisfy all of the client's requirements. Then I built the tasks to test that the classes were effectively fulfilling those requirements For example, the client required the id on the contacts to be unique. I implemented logic in the createContact method of the ContactService class that compares the id of the contact that is being created in the method to those of the existing contacts. If there is a match, the system throws an error. In my testing, I tested that I could create a contact with the createContact method using a unique id and that the createContact method throws an error when I try to create a contact with a duplicate id.

I can ensure that my JUnit tests for the ContactService and TaskService classes are high-quality because I was able to cover 100% of these classes in my tests. This means that all lines of code in these two classes were tested. I was very detailed in my testing to guarantee that all of the functionality of the service classes were tested properly. I also set up JUnit tests for the Contact and Task classes to reach 100% coverage there as well.

To ensure that my code was technically sound I tested various scenarios to make sure that all validations were working to meet the client's expectations. For example, in my TaskTest.java file on lines 80-88 I test that the setters are working when you enter parameters that meet the client's requirements. Then, I set up tests for the validations on each attribute specifically. For example, the testNameValidation test on lines 36-55 tests that neither the constructor nor the setter accepts a name input that is over 20 characters long, null, or empty. We also are testing that when we try to submit the invalid input, an IllegalArgumentException is returned, as opposed to an error message. This is an added level of detail that helps test more the behavior that we expect more thoroughly.

To ensure that my code was efficient I followed the general principle of removing duplication in code. For example, I realized that each attribute in the Task class would have to be validated for character length and existence to meet the client's requirements. This validation would have to happen in both the constructor and each setter method. Therefore, I was able to abstract that logic into one validation method called validateAttribute (lines 56-62). The method takes in an attribute parameter that represents the input value that is being validated and a character length limit. It evaluates the attribute to ensure that it is not null, not empty, and not over the character limit parameter. If one of these evaluates to true, the method returns false and the system will throw an IllegalArgumentException error.

Testing Techniques

I used the same testing technique for each milestone but edited the tests to ensure that each class fulfilled the requirements. For example, all classes required a unique id that was less than 10 characters long. For this attribute, I was able to reuse my tests for all 3 assignments. However, there were class-specific attributes such as 'name' for Tasks and 'appointmentDate' for Appointments. For these attributes, I kept the same tests but edited the attributes and testing requirements to meet the class's requirements. My general technique was to write functional unit tests until I had 100% coverage over the class that I was testing. This included making sure that all conditional statements behaved as expected with true and false results. I also had some integration tests for the service objects that were required for each milestone. For example, each service object required a way to edit the class that they are serving. His process relies on the getters and setters of the main class (Tasks, Contacts, and Appointments). I had to write integration tests between the service object and the main class to ensure that the service object is correctly modifying an instance of the main class.

There are many software testing techniques that I didn’t use based on the scale and the practicality of this project. To name a few:

- I didn’t use test-driven development (or TDD for short) in any of the milestones. TDD is a method where the developer writes tests for the desired functionality first, then codes the functionality until the tests pass.
- I didn’t conduct performance testing, which is when you test the application’s performance, responsiveness and stability when under a certain load.
- I didn’t do any security tests, which uncover security vulnerabilities in the application.
- I didn’t do usability testing because there is no front-end to our application. Usability testing is when a tester goes uses the application like a typical user to uncover bugs or other issues. This can be done where the tester knows the design of the application (called white box testing) or where the tester doesn’t know anything about how the application was designed (called black-box testing)

Each testing method discussed has a practical use and can be effective in the right situation. Here is a list of practical uses for each method highlighted in this reflection:

- Functional tests are great to apply to classes to ensure that their constructors, getter, setter, and helper methods all work as expected.
- Integration tests are useful when you have multiple components interacting in one application and you need to make sure that the functionality of a component that relies on another component behaves as expected.

- TDD is a great testing technique for most projects. It requires the developer to look at the big picture from the start and thoroughly think through the components needed and the way they'll interact. This is a technique that is better suited for experienced developers that are more comfortable with looking at the big picture.
- Performance testing is good for large applications with a lot of users. We need to understand how the application will hold up when put under stress. Users are prone to leave the site if it's taking too long to perform a task.
- Security tests should be done for any application that will be hosted on a server. Vulnerabilities in dependencies can lead to major data leaks. All dependencies must be tested often to keep up to date with the latest security updates.
- Black-box testing is useful to get a sense of potential roadblocks that users may face if they haven't used the application before. When the project team is focused on the application for months, it is hard to see the final product as a customer with no context at all. Black-box testing can uncover those weaknesses and should be used for applications with a wide variety of users.
- White-box testing should also be used for any application that is hosted on a server. The testers have in-depth knowledge of how the application was built and can make the debugging process faster for developers.