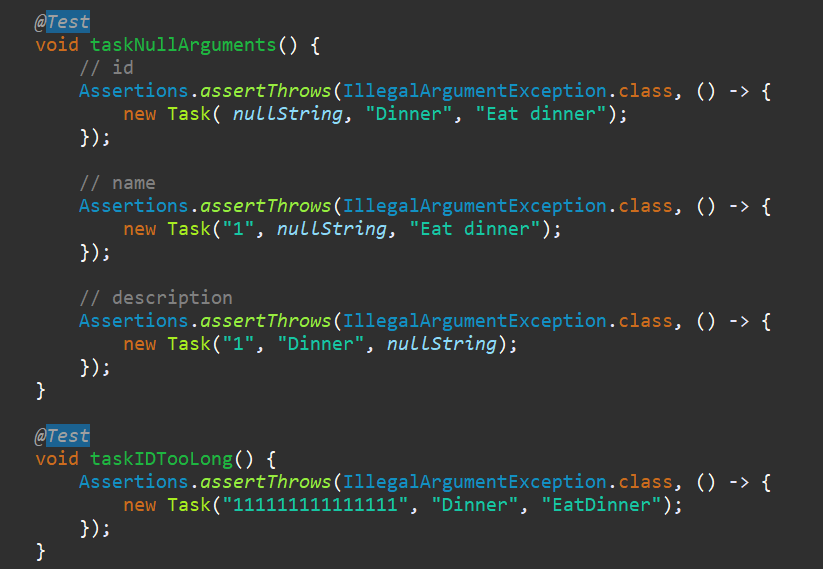
Jason O’Connell

10/10/2022

Project 2

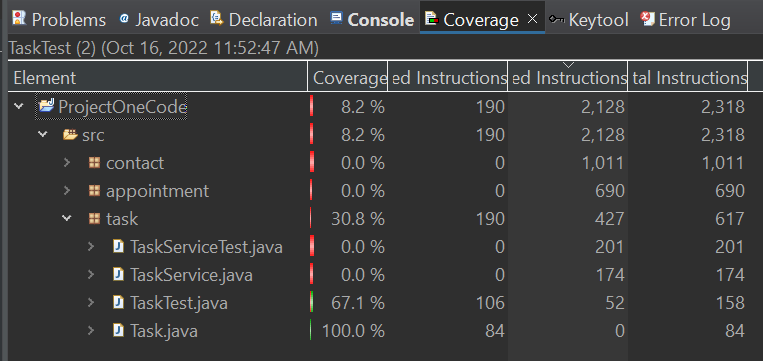
My approach in developing these three features was quite similar. The specific data for each differed slightly but the data types that each of the three components utilized were essentially the same. Additionally, the requirements for each of the components were uniform, with each having their own quirks. I paid special attention to the individual requirements for each of the three features. To keep with the uniformity of the requirements, the code across the three features is formatted in an identical way. This made for a readable, coherent, and reusable experience across the project. As you will see later in this document, the tests verify that the code accurately reflects the requirements.

One of the biggest requirements is checking a string for length and null string:

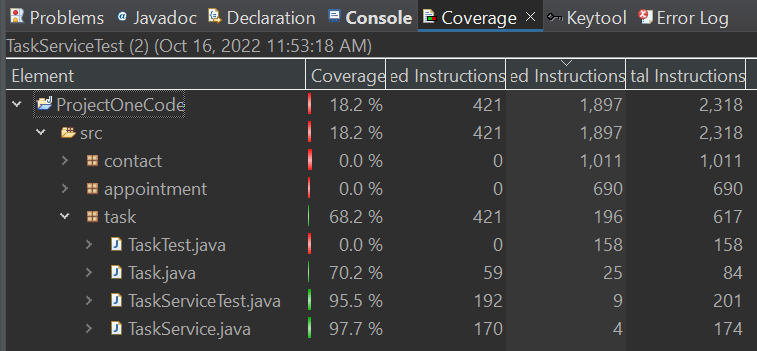
From the above image, you can see that the first test checks for an exception thrown with a null string provided. Also, you can see the bottom-most test check for an exception thrown with a string that is too long. This is only a fraction of the total tests implemented and each requirement is tested in this fashion. Further, the requirements for each feature were implemented as tailor-made tests to ensure functionality. For further evidence, please reference the source code.

The overall complexity of the code was fairly mild. This made testing very straightforward. Lower complexity means that there are less possible outcomes in terms of execution flow. Further, the testing amount does not need to be anything extravagant to get near-full code coverage. All of my tests were above 97% code coverage. 12% higher than the requirement.

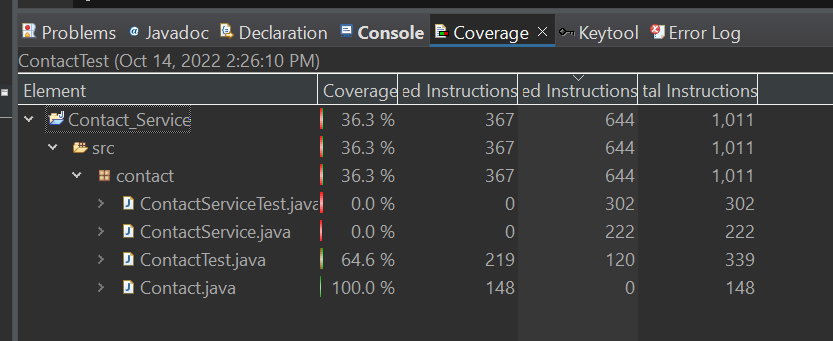
Task:



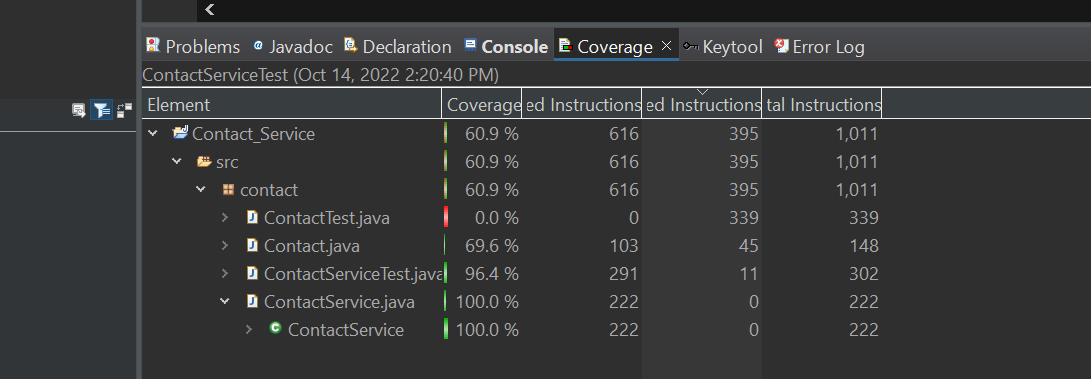
Task Service:



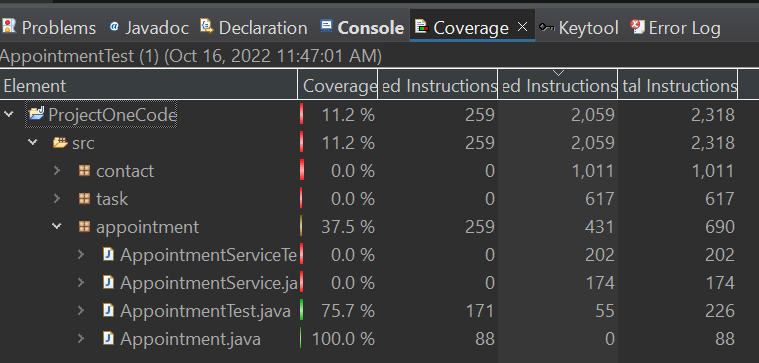
Contact:



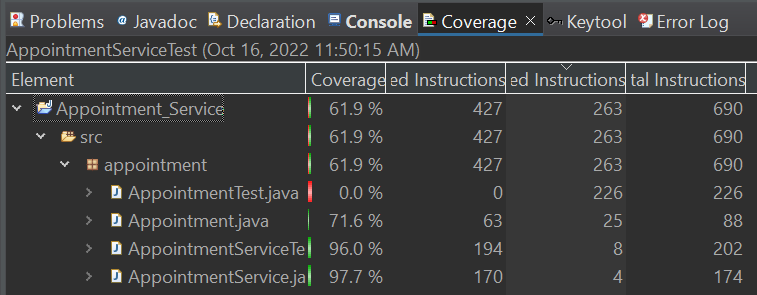
Contact Service:



Appointment:

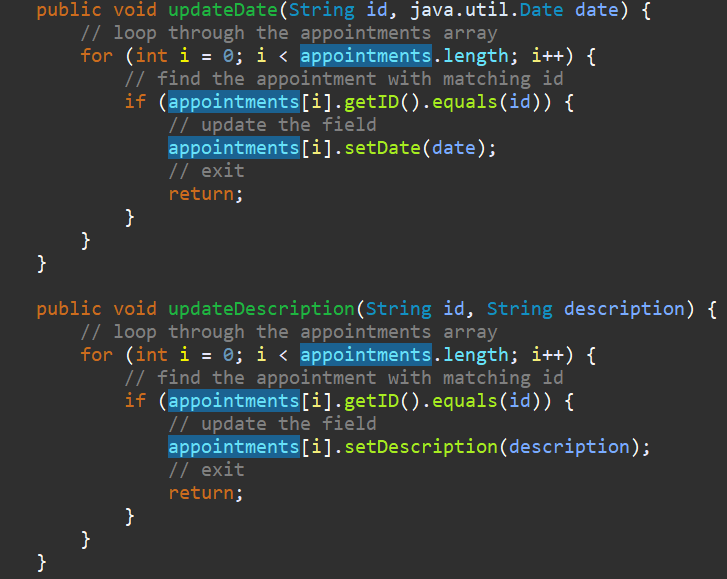


Appointment Service:



Testing coverage directly corelates to ensuring the technical validity of the developed code. If your test only covers 50% of the code, that’s a lot of room for error. With near-full code coverage as I had, the tests were able to validate the quality of my code and ensure that the functionality of the code performs as expected in a dynamic (non-static) environment. Not to mention, all of my tests passed which means that the code did what the tests required it to do.

This project did not have many avenues for efficiency. It is a straightforward application that was not very big. The bigger the application, the more avenues there are to explore efficiency. Part of the inherent efficiency of this application is the fact that it uses so few lines to accomplish its tasks. However, there is one area that comes to mind where I injected efficiency. See the example below:



Both of these functions are apart of the AppointmentService class. They attempt to update one of the appointments in the array of appointments stored by the AppointmentService instance. The methods are searching for an appointment instance in the array that matches the ID provided as a parameter. The code loops through the array checking each appointment instance. The efficiency comes in the “return;” statement. Without the return statement, this loop would continue even if it already found a matching appointment instance. This essentially means that the code would perform unnecessary iterations. The return statement prevents this. This efficiency addition has been injected throughout the project. Wherever this looping logic is, the efficiency “return;” statement was added.

The main testing strategy throughout the development process was unit testing. In our development, we built the three major components from scratch. When you are developing from scratch, it is good practice to develop unit tests alongside actual development. Once a particular method was built, we developed unit tests to verify that the method was behaving as intended with typical and atypical inputs. These methods formed classes which formed components. Each component consisted of a data class and a service class in which the service instance managed multiple data instances. Once the component was complete, we were able to perform some intra-component integration testing within unit tests.

We did not use any static testing tools to test this code beyond the built-in syntax checker. Using a static tester in eclipse is not overly difficult, but it does involve installing a 3rd party plugin. Many software projects will perform static testing to check for known vulnerabilities or errors. In larger projects with 3rd party dependencies, a static dependency check is performed to identify vulnerabilities in the 3rd party packages that the project is using. Zooming out, we did not perform some of the more broad-scope tests like acceptance testing or system testing. At this stage in development, the subsystems have not been assembled into a system, so the broad-scope tests would be premature at this juncture. Once the system is built and ready to be tested at scale, the system tests and acceptance tests can be performed. System testing in this context might look something like a run through of the different features. For example, the system testing might try adding and removing contacts, editing appointments, or viewing tasks. System testing should give the development team a sense of how well the system works in terms of subsystem cooperation. Acceptance testing is likely going to involve stakeholders. The stakeholders will typically define a list of “must be able to do” items that the system needs in order to ship. Acceptance testing in this context would look similar to system testing in that the stakeholders would want to see that the system can make appointments, create tasks, add and remove contacts...etc. The system performing these “must haves” in the presence of stakeholders is how the stakeholders know if they have a product that is ready to ship.

My mindset for this project was to be methodical and thorough. I could have easily rushed through everything, but that would not have allowed me to exercise caution. I exercised caution in this project by slowing down and thinking through what I was writing before I wrote it. For example, the mutator methods allow the client to set a value to an attribute, some of the attributes have certain requirements that were defined in the instructions. Omitting checks for these requirements in the mutator method is something that I very well could have overlooked had I rushed through the build process haphazardly. My methodical approach allowed me to see that the mutators needed to have input validators. In the example setName(String name) mutator method shown below, the inclusion of the validators can be seen.

public void setName(String name) {

if (name == null || name.length() > 20) {

throw new IllegalArgumentException("Invalid name");

}

this.name = name;

}

In adhering to good practice, I included specific unit tests that ensured that the validators in both the constructor and mutator functions were working as expected.

I tried to limit bias by building tests that were representative of circumstances that we needed to ensure requirements were met. An example of bias would be only including tests that represented typical interaction with the code. In order to eliminate bias, you need to test both typical and atypical circumstances. I believe I did well with eliminating bias. For example, I included tests that fed the code data that was of atypical circumstance. For instance, constructing a contact object with a null string name, which the code throws an error for. My tests showed that the code performs as expected under atypical circumstances. This is how I tried to eliminate bias.

Attention to detail is critical in both development and testing. As I mentioned previously, rushing development and/or testing is where mistakes happen. If your tests don’t catch it, that mistake is going to make its way to the end user. Shipping defective code is something that developers never want to do, especially when there’s risk of harm to people by using defective software. To limit the risk, it’s important for developers and testers to follow good practice and uphold standards set by the organization and the industry. I plan on making every effort to ensure the integrity of my production code and pointing out deficiencies as soon as they become evident. Then, taking the proper steps to remedy the situation is how I will maintain my reputation as a disciplined developer.