CS320 7-2 Project 2

Michael Ross

Department of Computer Science, Southern New Hampshire University

CS320: Software Test Automation and QA

Prof Ramsey Kraya

December 10th, 2022

[Summary 2](#_Toc155667083)

[Appointments 2](#_Toc1159562583)

[Contacts 3](#_Toc652897471)

[Tasks 5](#_Toc159029925)

[Reflection 6](#_Toc462502307)

[Testing Techniques 6](#_Toc1358601642)

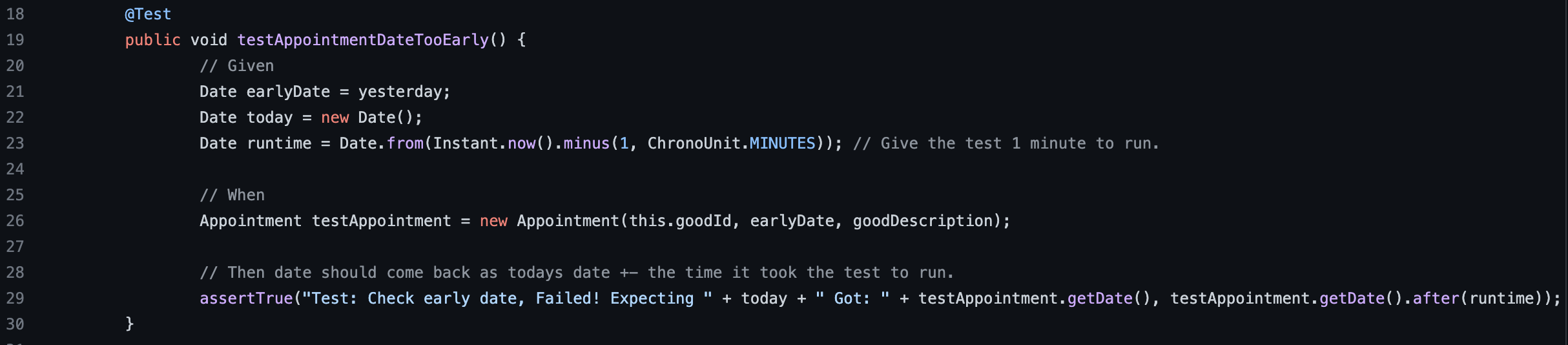
[Mindset 7](#_Toc1389098922)

# Summary

In unit testing the three classes developed for this project, the testing design was tightly coupled with the requirements. The requirements directly drove the design of the tests, and they were designed solely to meet their needs. Some of the tests were designed to meet the needs of the explicitly stated requirements, but there are others not explicitly stated but necessary to achieve the application's goals.

## Appointments

This class's requirements were around monitoring the input size to ensure they were under a specific length. There was one requirement for the date of the appointment which ensured that the date could not be in the past. Line 29 in the AppointmentTest class is where the test occurs.



This test accounted for the time the test took to run by adding an additional factor to the date element.

The testing of the length parameters involved the standard length testing process of testing the exact length and then lengths longer than the specified length to verify that the length was being truncated properly. This can be seen in all length testing performed for all classes created.

The third and final test for the Appointment class was testing when any of the values were passed as null. Since this was accounted for in the code, and the values were converted to safe alternatives, the tests passed well. The test involved attempting to set the value of any of the appointment variables to ‘null’ and then verifying they were not null in the object being tested. An example of this can be seen on lines 48 through 59 of the AppointmentTest class.



The AppointmentService class testing was less rigorous as many of the requirements had already been accounted for in the dependent Appointment class. The AppointmentService class needed to be a singleton, so testing creating a singleton was necessary. It also allowed testing the uniqueness of the ID field by generating two new Appointment objects in the AppointmentService class and verifying their IDs were unique. Since the size of the inputs has already been validated in the Appointment class, the other requirements to verify were the adding of new Appointments, deleting Appointments, and updating the fields of an existing appointment. These were done by adding a new Appointment object using the method in the AppointmentService class and then verifying that the new Appointment object exists. Similarly for testing deleting the Appointment object, a new Appointment object was created, verified, and then deleted to ensure that the deleted Appointment object no longer exists. This can be seen in lines 44 through 56 in the AppointmentServiceTest class.



## Contacts

The contact class requirements were entirely geared towards ensuring the inputs were not null and did not exceed a specified length. To test these requirements the standard procedure of testing inputs that were too long or exactly the right length was employed. In addition, null values were tested by setting values of the class variables to null. By verifying the expected output vs what was output, it can be shown that the intended result was achieved and fail the test if it was not achieved. This can be seen in lines 47 to 74 in the ContactTest class with the phone number length testing.



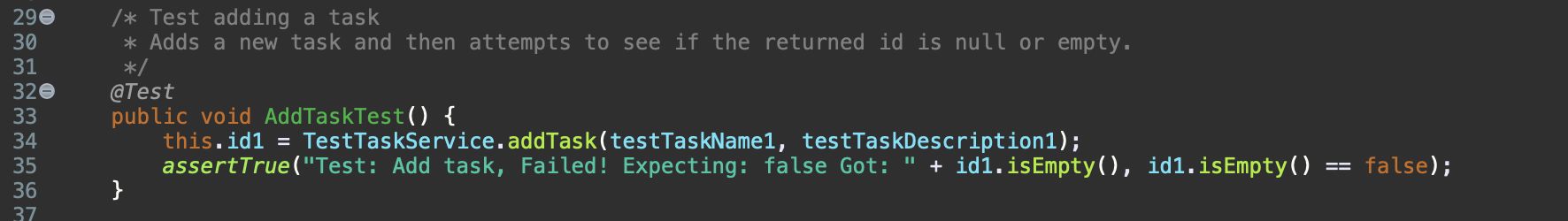
To test the ContactService class, the same procedure as for the AppointmentService class was used. A ContactService class was created and then contacts were added, deleted, and updated in that class to ensure that the contacts were successfully added, deleted, or updated. In addition, two contacts were created to ensure that the contacts had unique IDs. This can be seen in lines 63 through 74 of the ContactServiceTest class where two contacts are added, and the IDs compared to ensure they are not unique.



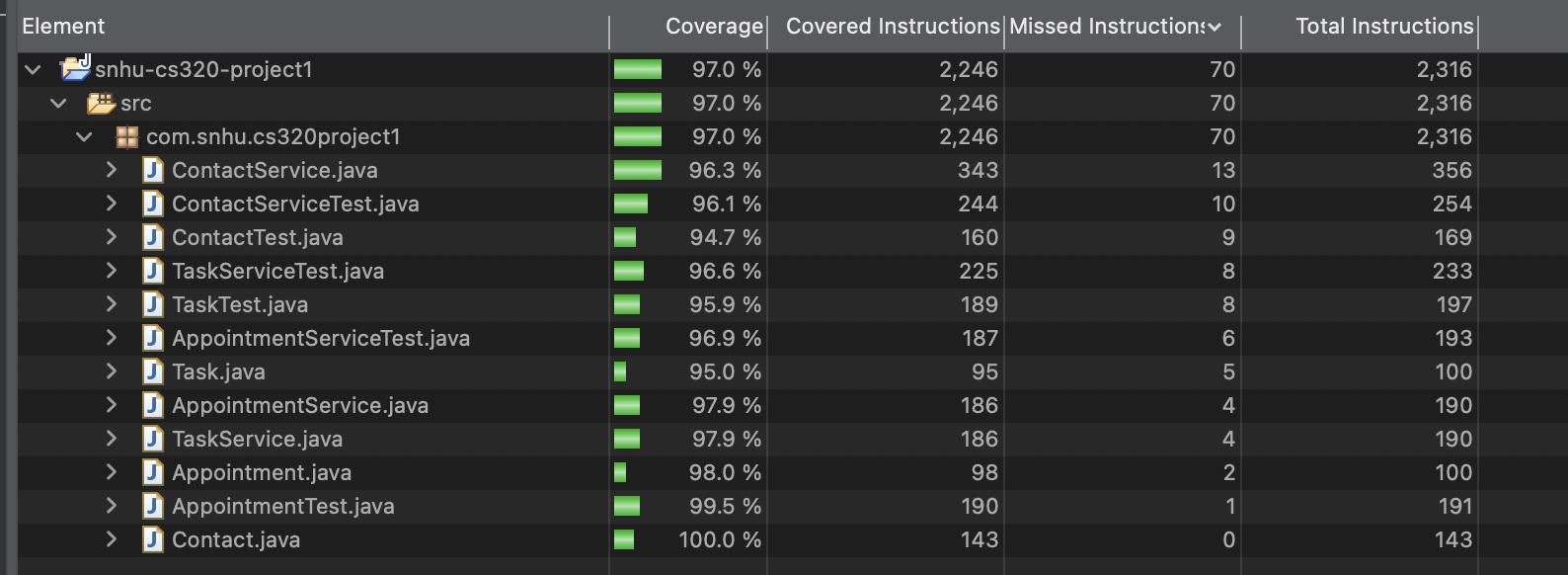
## Tasks

For the Tasks and TaskService classes the testing was the same as the previous two classes. The TaskTest class tests the lengths of all variables and sets them to values too long, the same length as the requirements specified and null. In this way it verifies all the requirements are met.

The TaskService class tests the standard requirements of ensuring the various Task class variables can be updated, ensuring that the IDs of each new Task added are unique, and that a Task can be added and deleted. The Task addition code can be seen in lines 29 through 36 in the TaskServiceTest class.



In examining the coverage for all three use cases, the coverage is above 90%, and in fact is above 95% for all covered code! This gives us great confidence that the code is being appropriately covered by tests.



The testing code generated to ensure the original code meets the requirements was created to be both efficient and effective. Standard testing methodologies were used to verify length requirements including testing for lengths greater than the spec, less than the spec, and the same as the spec. Examples of this can be seen in the screenshots above.

# Reflection

Most of the tests used in this project represent Black-Box testing methodology. It involves creating tests that validate the requirements without needing to know what the actual code is. Many testing techniques were used to verify the code specifics.

## Testing Techniques

Boundary value analysis (BVA) was used to verify the validity of the class variable lengths. As its name implies, BVA is used to test the boundaries of a particular item under test. Testing the boundaries is effective in meeting the requirements without having to test every single possible value. This involves testing things like variable lengths where the exact value of a variable length, a length under the exact value and a length over the exact value will be tested.

Decision Based Testing was used to verify the requirements were met in the various Service classes created. To test the deletion of the appointments, contacts and tasks, the Services objects needed to be configured to be in a specific state. With them in that state the deletion results could be tested appropriately. Decision Based Testing involves creating a table of states and then testing with the object in the various states to verify the outcomes.

One testing technique which was not utilized was a State Transition Diagram. None of the requirements necessitated taking the objects through various states to test the values at each state. With a more complicated set of classes or functionality, this type of technique might be needed.

For White-Box testing, code coverage was analyzed. This involves detecting how much of the code has been tested in the test cases. It utilizes functionality in JUnit to analyze the code automatically for coverage. The threshold set for this particular set of classes was 80% and the code coverage was 97%. See the screenshot above for the specifics.

Static testing was not utilized for this project as the proper tools were not available to us during the project. Static testing involves scanning the code using third party tools to verify whether the code conforms to coding best practices. The benefit of this type of technique is that the test can be run without compiling the code. This makes it valuable early in the software development process.

## Mindset

In acting as the software tester role as part of this project I tried to truly imagine myself in a real-world project. Caution was important as any bugs or issues which get through the testing process could mean loss of revenue or reputation for the company. Because I was also the one writing the code, I had a good appreciation for the complexity and relationships between the base Classes being created and the Service classes being created to manage the collections of the base classes. Most of the tests being run required a good understanding of how the code works to thoroughly test to the requirements laid out. Specifically, without understanding that the Service classes code needed to be a singleton, the tester would not have known to create a test for that possibility.

Bias can be a big issue in testing, especially when creating test cases which relate to classes of people. Since these classes do not deal in any data that could be biased, the possibility of bias is low. However, if there were a possibility of bias more care would have to be taken to ensure it did not taint the code.

Because the type of testing we were responsible for, unit testing, happens very early on in the coding process it has an oversized effect on the quality of the code. As has been shown in the past, the earlier a defect or bug is caught the less it costs to repair it. Being one of the earliest code testing techniques, Unit testing can have an enormous impact on the success or failure of a project. Because of this great care must be taken in ensuring the right amount of rigor and discipline is used in testing the code. For example, not testing this code to ensure that all IDs are unique could introduce difficult bugs in the future that could result in customer information overlap and serious security concerns.