

CS320 - Software Test Automation & QA

Dr. Morton

Brigitte Rollain

13 October 2023

Summary

Creating the three features of the application- Contact Services, Task Services, and Appointment Services- utilized an approach for unit testing that involved white box and black box testing. This was done to ensure the code aligned with the software requirements.

The white box testing helped find mistakes in the internal environment which helped to improve the quality of the code. A mistake that had happened in the Task object that employing white box testing had found was a subtle, yet crucial error in lines 73-75 of the Task Class:

```
“validateName(name);
```

```
this.taskId = name;” (Rollain, 2023), that is supposed to be in the getTaskName method in the Task class but was found to set the value to the task ID instead of the name. Before the test, the code seemed to be immaculate then this obvious error was found that had me reexamine the rest of the code to ensure no similar mistakes had been made. As well as quickly fixing the error to the current lines 73-75 of the Task class,
```

```
“validateName(name);
```

```
this.taskName = name;” (Rollain, 2023), showing that the task name is being assigned in this instance rather than the task ID.
```

While I was diligent in creating each feature- having no such error in the Contact Service- a similar mistake occurred in the Appointment Service feature. That was also found with white box testing and quickly dealt with accordingly. Utilizing white box testing helped maintain the internal code according to the requirements by assigning each variable, object, and list to their specifications.

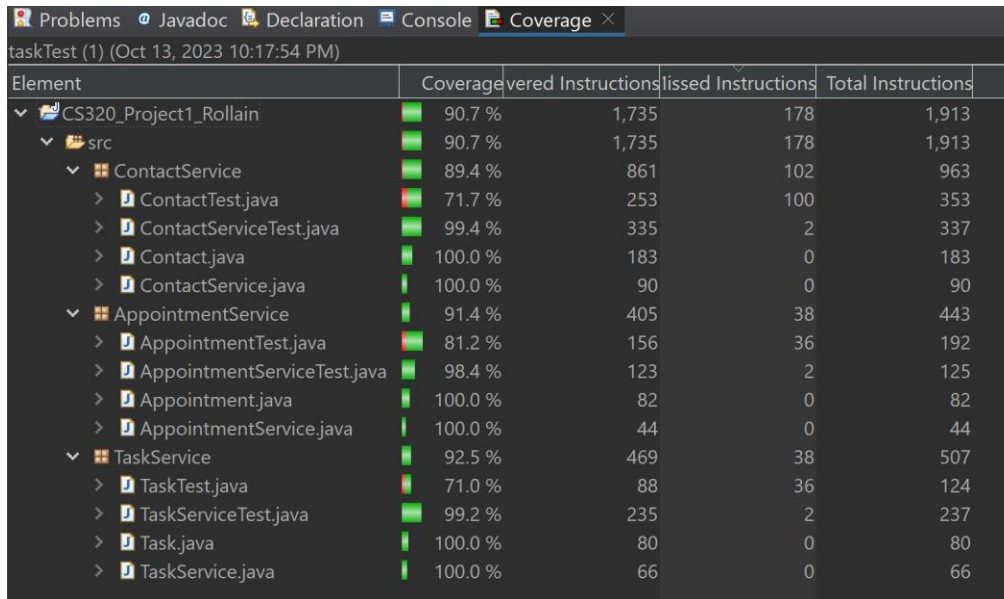
Black box testing ensured that the application would produce the correct results and was tested in each feature of the application in a similar manner, so I will only show the first instance that was created in the Contact Service test class (ContactServiceTest) test in lines 51-67,

```
51 • @Test
52 void testUpdateFirstName() {
53     ContactService cService = new ContactService();
54     assertTrue(cService.contactsList.isEmpty());
55
56     cService.addContact("Rick", "Sanchez", "5552347890", "426 Science Ave."); // Contact at Id 0
57     cService.addContact("Invader", "Zim", "1235557890", "909 Galactic St."); // Contact at Id 1
58     cService.addContact("Edgar", "Raven", "7895551234", "626 Nevermore Rd."); // Contact at Id 2
59
60     assertTrue(cService.contactsList.get(0).getFirstName().equals("Rick"));
61
62     cService.updateFirstName("0", "Beth");
63     assertTrue(cService.contactsList.get(0).getFirstName().equals("Beth"));
64
65     cService.contactsList.clear();
66
67 }
```

(Rollain, 2023).

The test first checks the instantiation of the list to be created and empty. Then adding contacts it checks the input to ensure it aligns with the order that contacts were entered. Afterward, the test utilizes a function to change the name value of the Contact object and promptly ensures that the change has been implemented by returning the expected value. As a side note, the Task and Appointment Service tests incorporate a size check to the list as an extra measure of quality assurance I thought of after the first iteration. Utilizing tests like this for each feature checks the output to ensure expected results and if there are unexpected results it will be examined by ensuring that the input and output match. The black box testing like this ensures that the output of the application is expected.

For the most effective testing, I strived for 100% coverage in each test for their respective class. This ended up getting me a total coverage of 90.7%, as you can see in the following picture showing the coverage results.



Element	Coverage	Covered Instructions	Missed Instructions	Total Instructions
CS320_Project1_Rollain	90.7 %	1,735	178	1,913
src	90.7 %	1,735	178	1,913
ContactService	89.4 %	861	102	963
ContactTest.java	71.7 %	253	100	353
ContactServiceTest.java	99.4 %	335	2	337
Contact.java	100.0 %	183	0	183
ContactService.java	100.0 %	90	0	90
AppointmentService	91.4 %	405	38	443
AppointmentTest.java	81.2 %	156	36	192
AppointmentServiceTest.java	98.4 %	123	2	125
Appointment.java	100.0 %	82	0	82
AppointmentService.java	100.0 %	44	0	44
TaskService	92.5 %	469	38	507
TaskTest.java	71.0 %	88	36	124
TaskServiceTest.java	99.2 %	235	2	237
Task.java	100.0 %	80	0	80
TaskService.java	100.0 %	66	0	66

(Rollain, 2023).

This picture shows the overall coverage at the top with the project folder that contains all the features. Looking at the different features it is shown that Contact Service has the lowest coverage at 89.4%, then Appointment Service with 91.4%, and Task Service with 92.5% test coverage. However, delving deeper it shows that the Contact, Contact Service, Appointment, Appointment Service, Task, and Task Service classes have 100% coverage from the tests. This means all the functionality of the application has been tested to align with the specifications.

Reflection

Several software testing techniques were employed in this project including - manual, automated, functional, unit, integration, acceptance, and boundary value testing. Manual testing involved a physical overview of the code that did catch some obvious errors that helped the code quality. The automated testing involved writing JUnit tests to execute written tests and found more errors that the manual testing did not find. The functional testing employed in the project verified the

code against the requirements for the project by implementing automatic tests that checked the ability of the code to match the expectations set for it in the tests. Unit testing has been described before as using both white box and black box testing to test both internal code and the application's function. When going from the Service class to the Object class, integration testing helped ensure seamless cooperation between them by checking the Object and the Service instance with each other. Acceptance testing is done to check that the application is understandable for end-users by making sure some functions are properly guarded and that what isn't guarded is useful for them to use. The Boundary Value testing is utilized to confirm that errors will occur when trying to input values that go out of the expected input values.

Some of the software testing techniques that were not incorporated into this project include- non-functional, performance, security, and usability testing. As the application did not use an outside source for information holding and no server, there was no need to test the most common non-functional uses for the application along with scalability. Performance tests weren't utilized for this project as the speed and scalability of the code were not a requirement for this project. Security testing was not done in this case as vulnerabilities in the code were found through other testing methods. While these testing techniques are typically encouraged to be incorporated into projects, there were no requirements that specifically called for these tests to be executed in the project.

The mindset for this project was not so much in caution as it was more focused on the experimental and experiential focus to experience what it is like to have various different errors, from typing errors to calling incorrect methods to examine the outcome of running the tests. Caution was employed to ensure tests were executed as expected, even when purposefully checking defects. Appreciating the complexity of the code through exposure to an environment that was not perfect helped me comprehend what happens when a defect is found and how to fix

it. In creating Task and Appointment Service classes I purposely coded as hastily as I could in the hope of creating errors to experience in conjunction with the immaculately coded Contact and Contact Service classes. Lines 28-32 in the Appointment Service class,

```
28 public void deleteAppointment(String aDesc) {  
29     int intId = Integer.parseInt(aDesc);  
30  
31     apptsList.remove(intId);  
32 }
```

(Rollain, 2023). This

picture shows an instance of hasty coding done in the first iteration of the Appointment Service. This shows the importance of making sure that the correct variable is being used in a method, otherwise, in this case, the value is unable to be converted to an integer thus, the result is unable to find and remove the corresponding value that will be an appointment in the list of appointments.

Bias was easy to remain level during this project due to an overwhelming curiosity about how automatic testing works to find defects within code. Due to this strong curiosity, I wasn't concerned about being told that the code I wrote was not precise. Despite my lowered bias, I can understand why a software developer may feel attacked when told their code has defects as programming for hours with manual tests between iterations feels like their worth as a programmer is being attacked. If testing my own code, I would feel some embarrassment in a professional environment to find code defects, but I would be more relieved to have found something through testing as that could be fixed.

Being disciplined does not mean being perfect, but rather acknowledging and moving past mistakes with determination. Quality code comes not from the lack of mistakes but from the finding and fixing of them. So, it is vital to not cut corners in both development and testing code to help create the most practical solution. To avoid technical debt, the reasoning for a function along with understanding how a function will be used in an application should be understood.

Another way to help reduce technical debt is to create tests as one develops their program to check previous and current developmental phases of a project and confirm that integration is being done smoothly.

References

Rollain B. (2023 September 30). CS320_Project1_Rollain.