Dathan Pompa

Cs 320 Software Test, Automation

6/30/2024

Project 7-2

Contact:

For my contact service class, I implemented CRUD operations for the contact records. This means it will create, read, update and delete accordingly. I tested the contact service class to align with the software requirements, but in my contact class I only tested for null. I needed to test if the given data value is over the maximum allowable characters. I did test these in my previous assignment but forgot to add the functionality to my project. In my unit test for my contact service class I should have asserted utilizing my getContact method instead of checking a return value, this will increase my code coverage as well as keep my code from having bugs later down the software development life cycle. My coverage percentage was high for my service class but scored 42% for my contact class, I believe the reason was because I created error handling within my object creation as well as within my methods. My unit test was technically sound because of my object creation and beneath it testing to see if the return value of the given object was inserted “ Contact contact = new Contact("1", "John", "Doe", "1234567890", "123 Main St");

assertTrue(service.addContact(contact));

assertFalse(service.addContact(contact));”

My test to verify null cannot be passed into the contacts for the ID is technically sound because of my object creation and making a test method to test if the tested data will allow a null value,

“@Test

public void testContactInvalidID() {

assertThrows(IllegalArgumentException.class, () -> {

new Contact(null, "John", "Doe", "1234567890", "123 Main St");

});

}”

My code is effective in a way that the object gets created and gets tested for certain values right after its creation, I do think I could cut back on reusing the same code twice and utilizing assertAll instead of asserting one by one to increase my efficiency by only handling Assertion Errors, it also allows us to test all asserts instead of having the program stop if one assertion fails and having to test the program again to see if others pass.

Task:

For my task test, I made a similar mistake by creating error handling within my object creation and inside of my methods, this is double the code that is being run and can be reduced to increase efficiency. It aligns with the software requirements by only allowing valid data to be given and those that are not null. My if statement checks whether data given is invalid if null or greater than the maximum length of the software requirements. Like my contact tests, I created multiple asserts which do not help with longevity or effectiveness. I tested that my task service class operates as intended by verifying the data aligns with the software requirements.

“@Test

```
public void testMakeTask() {

Task task = new Task("1", "Task1", "I am talking");

assertEquals("1", task.getTaskId());

assertEquals("Task1", task.getName());

assertEquals("I am talking", task.getDescription());
```

}" As you can see, I used multiple asserts, I should have used assertAll so that all asserts can be tested and it would not stop the program from running because an individual assert failed the check, requiring another run of the program. My task test coverage scored 82%, but my task class scored 52% because I doubled the amount of error handling needed by adding error handling in my methods when it was already done in my object creation. My code is technically sound because I create an object and test to see if the data was successfully passed into my program, but improvements can be made.

Appointment:

My appointment class aligns with the software requirements by having the error handling within the object creation, but I added double the amount of needed error handling in my methods when I did not need to do so. Checking code coverage helps show what code does not run and which does which can help you change your code to be more efficient. I successfully tested my service class by creating an object and testing if the data was successfully passed into the object and created asserts right below the object creation. I can increase the usability of my code by using assertAll instead of individual asserts, this is because if one of the asserts fails, I need to rerun the program to see if the other asserts, after the one that failed, will pass the test. My

appointment test unit test has a coverage score of 60% because of my testAppointmentInvalidID method, it has 0 coverage compared to my test make appointment method. “@Test

```
public void testMakeAppointment() {
Date futureDate = new Date(new Date().getTime() + 86413457);
Appointment appointment = new Appointment("1", futureDate, "I am talking");
assertEquals("1", appointment.getAppointmentId());
assertEquals(futureDate, appointment.getAppointmentDate());
assertEquals("I am talking", appointment.getDescription());
}

@Test
public void testAppointmentInvalidID() {
Date futureDate = new Date(new Date().getTime() + 86412345);
assertThrows(IllegalArgumentException.class, () -> {
new Appointment(null, futureDate, "I am talking");
});
```

}” My code is technically sound because I successfully created an appointment object and tested if the given data was valid, I could have increased the efficiency of my code if I added another asserts to test if a new appointment object will accept a null ID in the same method instead of creating another one. I can also use the assertAll to test all asserts instead of individual

asserts to save myself time from running my code over and over again. My appointmentService class is efficient because I create an object and test if the data created is valid right under the object, I can make my code more effective by using assertAll.

Reflection:

Some techniques I employed in my unit testing are Boundary analysis, test coverage and detecting anti-patterns within my code. I tested the limits of my data inputting maximum accepted data into my object pushing my code to its limit, hoping to trigger an edge case if caught. I used test coverage to see where code could be unused and attempted to find ways to make my code more useful. I was able to find anti-patterns after a code review from my friend, he showed me where the input data was going and how certain error handling, I implemented was unnecessary. Some unused techniques that would be interesting to employ in my code would be integration testing, user acceptance testing and security testing. Integration testing would be useful to employ because of how many classes are working together, seeing how my data could be used to test if all classes work in synchrony would aid in my learning. I would need to prepare a database first to deploy an integration test, but it would be a fun project to do. Another technique is user acceptance testing. This technique is fascinating because this is where all great projects gather important data from end users. This is when code gets pushed into the Beta phase and requires real time users interacting with it to test if the software works or not. This is an important stage to employ because it helps ensure user needs and expectations are met. Testing my codes security is also something important to employ because it helps us find any vulnerabilities within my program and allows us the vision to correct how we handle our data. Solving such issues could be done with best coding practices, such as encapsulation and obfuscating data being transferred so that it cannot be read if found. My mindset I used going into this project was to be cautious, making sure the input data is aligned with the software

requirements. I would test the limits of the input values to make sure that the software ran as intended, but I created too many unused error handling within my classes. I think if I implemented white-box testing while creating my code, I could have avoided this issue. That is because I would have made sure that I was passing the data correctly through my program instead of double-checking. I should have utilized a limiting bias mindset because this would have made me aware of my mistake. Having another developer review my work helps me to reframe my perspective of the goal. I unfortunately did not try to limit my bias in my review, after submission. I did, however, ask a friend to look over my code and we had a very good discussion about what went right and wrong. I have my own plan to carry out when creating code, but seeing a more experienced developers mindset helps me improve the quality of my code and reduce the time it takes me to think about ways to solve problems. It is important not to cut corners when it comes to quality code because it will save a company time and money later down the SDLC. Having technical debt is tough to fix especially if the life of a program is old, fixing issues early on helps teams not worry about whether a program will fail if they try to create modular code that interacts with the code that has bugs. Cutting out technical debt is achievable by making sure my testing practices are thorough, well defined and limiting my bias by reviews from peers. When I cut corners by not having a peer review, I suffered by having low code coverage and less efficient code by not utilizing assertAll. My friend was able to spot problems they have made in the past quickly and the advice given to me would have allowed me to score better on my project submission.

Citations

Stefan Bechtold, S. B. (n.d.). *2.5 Assertions*. JUnit 5 user guide. https://junit.org/junit5/docs/snapshot/user-guide/

Morgan, P., Samaroo, A., Thompson, G., Williams, P., & Hambling, B. (2019). 4 Test Techniques. In *Software Testing : An ISTQB-BCS Certified Tester Foundation Guide* (4th ed., pp. 91–155). essay, BCS Learning & Development Limited.