**7-2 Project Two**

Tomas J. Estanislao

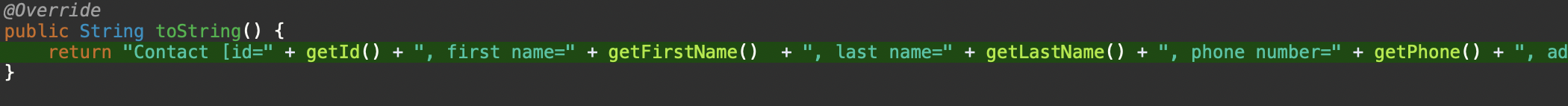
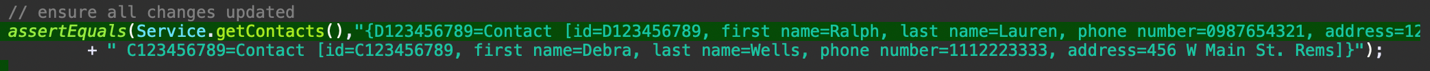
Southern New Hampshire University

CS 320: Software Testing, Automation, and Quality Assurance

Professor Kalysa Wilson

December 16, 2023

**7-2 Project Two**

For all services, it was important to take notice of both the required fields and the required methods. Not only is it crucial to understand specific customer requirements, but it is necessary to understand how each required field interacts with each other. These interactions take place both within each object, and from the respective service that implements it. I will describe my testing approach for the contact feature, as I follow the same process to implement the rest. Due to the architecture of this application, it was necessary to implement the contact class and write the JUnit tests thereafter in ContactTest— before designing the contact service. My approach was to have 5 separate tests for each required contact variable, and an additional test to confirm what a successful validated contact object looks like. The testValidContact() test confirmed string lengths within the designated character limit. For each of the subsequent tests, I needed to confirm that invalid inputs would be appropriately rejected by my program and returned with an illegal argument exception. I tested for both the max string length of an argument and above. I also confirmed that entering in a null value would not be accepted. These were the main points that I focused on for my ContactTest JUnit file. Similarly, I had to implement the ContactService before writing JUnit tests. I did focus on the 3 main requirements for adding, deleting, and updating contacts— which translated to 3 tests. I needed to test for the required functionality of each method, but I also needed to confirm that the contact class requirements were also properly addressed. Specifically, the Contact ID was a unique required field for the object, “that shall not be null and shall not be updatable.” I needed to confirm in my tests that my data structure was properly implemented, so that a duplicate ID would be properly rejected. I also had to confirm that each method works in isolation; you should not be able to update a contact through the add method, and you should not be able to add a contact through the update method. Lastly, I verified that illegal arguments were rejected by designing a getContacts() helper function to retrieve the total contacts in memory, as well as an overloaded toString() function to display the contents of each contact object. A screen shot of a computer program

Description automatically generated

Because the appointment feature largely followed the same format as the other features’, I was able to duplicate and refractor each object and service based on the previous two. In the requirements, neither date nor description said, "shall not be updatable." Therefore, I decided to allow updating by fields, and used chatGPT to help generate code for random Date testing— as I was unfamiliar with the outdated Java Date class. I ended up implementing an additional update method to the appointment service.

A computer screen with colorful text

Description automatically generated

Coverage percentage was an indicator that I used, not only to determine if my code was being effectively tested across the board, but to assist in narrowing down bugs as well. For instance, there were times when if/else branches were not executed, and without the coverage percentage I would have had a harder time narrowing down why tests were failing. Because of this, I was able to revise both my conditionals and branches. Coverage also helps in determining any unnecessary code, as well as checking that helper functions are properly implemented and utilized in the tests. I also made sure to create a test suite that could run both test classes at the same time, to get an accurate measurement of coverage for the program. For the final project, consolidated all three test suites into one master suite to run every test.A screen shot of a graph

Description automatically generated

I will use my testID() test as an example for technically sound code. I used assertions to check that my exceptions were thrown correctly, and I did this for illegal inputs that broke the software requirements. Although I set my ID variable to private; I did not implement a setter. So, I wanted to try mutating the ID within the test to validate that no changes were made. This was one example of ensuring that my code was technically sound, and met all of the software requirements for this required field. Writing tests for Date objects was a bit tricky. Not only did I need to test for invalid past and null Dates, but I also needed to implement a random future and past date to use for my boundary tests.

A screen shot of a computer program

Description automatically generatedA screenshot of a computer program

Description automatically generated

A computer screen shot of a program

Description automatically generated

A screenshot of a computer

Description automatically generatedStoring and accessing objects was the primary purpose of each service listed in the software customer requirements. So, I understood that I needed a fast and easy way to both retrieve IDs and restrict uniqueness. A HashMap conveniently allows for quick indexing by unique keys. This data structure fulfills many of the functions necessary for efficient local storage and pre-defined methods like: clear, get, remove, and put. Testing wise, I will use testAddContact() as an example. I tested that a contact was properly added to the collection. I also made sure that both duplicate contacts and IDs were rejected by the service. Lastly, I validated multiple successful additions to the collection. For my addContact() function, I nested my contact creation code within a conditional, ensuring that the contact didn’t already exist. I also let my Contact object take care of validation of parameters, rather than redefining new validation in the service class. This is an example of efficient code that uses the advantages of data structures that optimize time and space complexity. My code is modular for easy editing and readability that avoids unnecessary operations and bloat. A screenshot of a computer program

Description automatically generated

In each milestone, my primary task was implementing unit tests. The techniques and their descriptions are referenced from *geeksforgeeks.com.* Unit testing is the testing of individual components of a software application. Concurrently, these tests look at the functional requirements of each class object and service. This is a form of automated testing; I use the JUnit5 library with ServiceTestSuite.java to automate the process of executing tests. I also check the state of each service after making modifications— which is part of regression testing. To ensure that each functional requirement is met, I boundary test each input value. For each parameter and service method, I test: the maximum allowed length, out of bounds, and illegal argument types— this applies to both String and Date objects.

I did not perform integration testing. This might have required implementing a phone or social media app— with profiles that use each contact, task, and appointment. There is also the possibility of creating tasks, that contain appointments, that contain contacts: like a nested linked list. I did not use any exploratory testing techniques, which is trying to find defects randomly. Non-functional testing— including usability, performance, and acceptance— was not applicable to this project. Implementing a front-end with a RESTful architecture would have been necessary to use the aforementioned techniques.

In an article on mobile app testing from *testilo*, “QA teams use manual testing to conduct exploratory testing that requires human-led insights to identify more nuanced shortcomings of the application experience... Mobile app product teams should also seek to capitalize on the efficiency and cost benefits of automation for the test case types that are appropriate. That’s why many enterprises leverage quality engineering services to build and manage automation test suites” (Gilmore). Manual testing is better performed with access to real data. Real user data at scale can provide unique insight to new testcases which may not have been previously accounted for. On the other hand, automation services can provide a unified testing framework with end-to-end processes. An article from *QAble*(2023) provides insight to use cases pertaining to game testing, “An application should perform as per the expectations of end-users. There are various versions of devices even in a single technology, and hence it demands compatibility testing. A gaming application should work with the same frequency across all mobile devices.” Compatibility testing would include things such as responsive design. The application should conform the dimensions of a users’ device and perform to their expectations. Testing on all supported operating systems, devices, and web browsers should be conducted. Like automation services, this can be conducted through virtualization software and services such as Docker.

The interrelationships between each object and service required caution— ensuring that each service was able to manipulate objects efficiently and accurately in a data structure. One example of an interrelationship, one I previously described, is validation. Since each object contains specific fields, that each have their own individual restrictions, validation needs to be implemented at that local level. However, each service also has the functionality to add, delete, and update objects. Because of this, it becomes necessary to re-test method inputs at the service level. To reduce overall complexity, the service can create new objects and update fields, while validation can be handled within the object— with all associated error handling therein.

A screenshot of a computer program

Description automatically generated

A screen shot of a computer code

Description automatically generated

A screen shot of a computer program

Description automatically generated

An example of bias I tried being aware of, and avoid, was congruence bias. From *softwaretestinghelp*, congruence bias is “a behaviour where our mind refuses to think of alternatives” (Geethanjali, 2023). One of the ways I did this was by conducting negative testing. Take the Date field as an example, I created tests for inputs consisting of a null and past date. In the corresponding appointment service, I also tested for those same negative inputs. This pattern continues for the ID field, “The appointment object shall have a required unique appointment ID String that cannot be longer than 10 characters. The appointment ID shall not be null and shall not be updatable” (SNHU). There are a few restrictions implied in these statements: ID’s with lengths greater than 10 and null should return exceptions, retrieving and mutating the reference to the ID should not change the stored value, adding a duplicate ID should result in no change, deleting an ID that doesn’t exist should result in no change, deleting the correct ID should return the updated container of appointments, and updating the wrong ID should result in no change. These are examples of cases where I had to go back and refractor my code throughout my entire project to ensure that I was properly accounting for my bias.

A standard I tried to follow and maintain was proper commenting. I described what methods and helper functions each service contained— in addition to providing updates on the decisions I made and code coverage. Within each method itself, I provided a brief description of what each method does, as well as any parameters it takes and the expected content it returns. In this scenario, cutting corners means not committing to a consistent documentation standard. This makes it extremely difficult both to write and test your code, as it is very easy to call on methods with improper inputs, as well as expect the wrong results in testing. In technical debt, this slows down the individual developer’s workflow with interrupting errors and confusion. It also makes it much more difficult for any team members to understand, debug, and refractor existing code. Another point not often mentioned is any non-technical staff that may review your code. Both from the perspective of QA, as well as management, it’s important to have readable code with descriptive comments— that can ensure proper testing and communication.

**A screen shot of a computer program

Description automatically generatedA screenshot of a computer program

Description automatically generated**

**References**

Geethanjali, S. (2023, Novemeber 27). *Cognitive Bias In Software Testing: Why Do Testers Miss Bugs?* SoftwareTestingHelp. <https://www.softwaretestinghelp.com/cognitive-bias-in-software-testing/>

Gilmore, L. *10 mobile testing types and approaches*. testlio. <https://testlio.com/blog/10-mobile-testing-types/>

Neeru360. (2023, November 24). *Software Testing Techniques*. GeeksforGeeks. <https://www.geeksforgeeks.org/software-testing-techniques/>

QAble. (2023, November 22). *7 Different Types of Game Testing Techniques*. https://www.qable.io/blog/7-different-types-of-game-testing-techniques