Summary and Reflections Report

Project Two

Nicholas R. Enquist

Southern New Hampshire University

Summary and Reflections Report

I was recently tasked with implementing, and testing, multiple features for a mobile application. These features included the contact, task and appointment classes, and their associated services. I implemented these features and conducted static testing through JUnit to ensure they met their requirements.

The first feature to be implemented was the Contact functionality. The Contact class has several attributes that have specific requirements in regards to their contents. To ensure these requirements were met, I tested the contactID, firstName, lastName, phone and address attributes to ensure that none of these were null, or exceeded a set length of characters. The ContactService class served as the interface for the Contact class. This class maintains an array of Contact objects, and allows the user to add and remove contacts via their contactID attribute, as well as updating the remaining Contact attributes. Testing of the ContactService class ensured that all methods properly looked up contactIDs, allowing further execution of the method’s functionality. The update methods call setter methods from the Contact class, each of which contains data validation to ensure all values stay within the set requirements.

The second feature was the Task class, and its service interface. Like the Contact feature, the Task class had several attributes with content requirements. In this case, the taskID, taskName and taskDescription attributes could not be null, nor could they be longer than a set character length. Through my JUnit tests, I ensured that all of these requirements would be enforced. The service class maintains an array of Tasks, and allows a user to add or remove Tasks by their taskID attribute, as well as update the taskName and taskDescription attributes. Further testing ensures that these methods validated the taskID values properly, while data validation within the Task class’s setters ensured the integrity of its attributes.

The final feature was the Appointment functionality. Similar to the previous features, the Appointment feature had several attributes, apptID, apptDate and apptDescription, that had requirements that must be met. None of these attributes could be null, nor could the apptID or apptDescription be longer than a set character length. The apptDate attribute had a unique requirement, as it used a date value, and could not contain a date that occurred in the past. These requirements were validated through static JUnit tests. The AppointmentService class also manages an array of Appointment objects. This service class enables the user to add or remove Appointments from this array, using the apptID attribute. This class was tested to ensure apptIDs were properly validated.

Total code coverage for my JUnit tests was approximately 77%. While my test cases clearly could have been slightly more efficient, they covered a vast majority of the code. This effectively provides an ample snapshot of the quality of the code’s functionality in regards to the client’s requirements.

To ensure my code was efficient, I consistently iterated across my methods, and refactored as necessary, to remove unnecessary complexity and interdependencies. Streamlining the code into the simplest, and most efficient, form it could take was the most important step in keeping the functionality at the forefront of the development process. This included cutting many lines from the ContactService class’s update methods, which, originally, had all of the data validation logic built into them, while also being built into the Contact class’s methods. This was extremely redundant, and causes unnecessary code complexity. I then implemented JUnit tests to ensure that each element of the Contact and ContactService classes were thoroughly tested to meet all of the client’s requirements, as collected at the beginning of the project.

All of the above testing was completed using static testing methods. Automated static testing, also referred to as static analysis, uses tools to look for defects without executing code (Hambling, et al., 2015). Static testing is performed by writing additional code specifically aimed at ensuring code performs its functionality within expectations, without requiring actual program execution. Static tests validate that methods perform within a specific requirement, such as ensuring that a variable falls within a set character length, or throws an exception, if it does not.

This project did not rely on dynamic testing, which occurs during runtime execution. During dynamic testing, the program is executed and tested using a set of test data, verifying that outputs and behaviors fall within expectations at runtime (Hambling, et al., 2015). For a truly comprehensive testing cycle, both static and dynamic testing should be utilized, if the project’s resources allow.

When I first began Project One, I went in with the mindset of a simple, functional developer. I began writing my code as I would have in previous projects, where input validation was handled through a series of loops and validation checks, rather than allowing the program to throw an exception. As I began to write tests, I realized the folly of this approach, as it became impossible to get any true penetration into my code. I then refactored everything I had written to make it more testable, moving most of the data validation into the storage classes (Contact, Task and Appointment), and out of the respective service classes. This allowed for much greater test penetration; what was once around 32% coverage eventually became 77% coverage. This new approach allowed me to better grasp the importance of reducing complexity and interdependencies within my code, whenever possible. Writing code to simply fulfill the goal of “it works” can lead to some extremely bad habits.

Writing with the intent to test brought more discipline to the development process. While it is impossible to completely remove bias when it comes to your own code (“I wrote that, I *know* it works!”), I happen to be my own strictest critic when it comes to my work and art. This heavy-handed approach to self-reviews has allowed me to improve my practices quickly and efficiently, as my primary goal is constant improvement. Learning to write code that is meant to be tested has helped me improve the way I organize both my code, and my thought process as I design the code before I begin writing. With this refactored mindset, I believe I’m in an even better position to avoid technical debt when I finally take a position in the industry. This improved outlook on the development process will allow me to be more critical about the design decisions that I make, which can lead to fewer defects and wasted time refactoring erroneous code.

References

Hambling, Brian, Morgan, Peter, Samaroo, Angelina, Thompson, Geoff, Williams, Peter. (2015). Software Testing - An ISTQB-BCS Certified Tester Foundation Guide (3rd Edition) - 3.4 Static Analysis by Tools. BCS The Chartered Institute for IT. Retrieved from

https://app.knovel.com/hotlink/pdf/id:kt00UC2IL5/software-testing-an-istqb/static-tes-analysis-by