Quintin B. Rozelle

8/7/2023

7-2 Project Two

CS-320

**To what extent was your testing approach aligned to the software requirements?**

I believe that my testing approach was very well aligned with the requirements. As an example, the Contact class requires five different variables with length requirements for each. In my ContactTest class, I created a test for each of these requirements to assess the Contact classes ability to adhere to these. The screenshot below shows the tests for the phone variable (not null, not less than 10 characters, and not greater than 10 characters). Additionally, the remainder of the tests used on the other variables can be seen on the right side of the screenshot. All function similarly to the tests for the phone variable.

A screen shot of a computer

Description automatically generated

**Defend the overall quality of your JUnit tests for the contact service and task service.**

The JUnit tests that were written for these classes achieved 100% coverage as can be seen in the following screenshots. This shows that the quality of the tests was high.

Contact and ContactService classes

A screenshot of a computer program

Description automatically generated

Task and TaskService classes

A screenshot of a computer

Description automatically generated

**How did you ensure that your code was technically sound?**

I ensured that my code was technically sound by adhering to generally accepted conventions while writing my code and ensuring consistency throughout. For instance, my method to update a task (see screenshot below) is named appropriately and descriptively, uses descriptive local variables, and calls methods from the objects it interacts with to do the work instead of having that re-coded again. In short, it adheres to the general principles of object-oriented programming and DRY programming.

As another example, I took the feedback from Module 3 and defined my taskMap variable as a Map instead of a HashMap. This increases flexibility in the code by allowing the taskMap variable to hold any object that uses the Map interface instead of only a HashMap.

A screen shot of a computer program

Description automatically generated

I do think it could be argued that aspects could have some improvement in being more technically sound. My validation methods (Task’s validateTaskID method shown below as an example) intentionally throw exceptions when the input doesn’t meet the requirements instead of dealing with it in that moment (e.g., asking the user to resubmit correct input, modifying input so it’s correct based on simple rules, etc.). From my understanding, it is generally not considered to be the best programming practice to intentionally code to an exception when logic could be coded instead. Despite this, I chose to use this route for these milestones as the intent was to assess the use of JUnit and not the ability to deal with incorrect input. Coding by exception was intentionally used for simplicity’s sake so JUnit tests could be focused upon.

A screen shot of a computer

Description automatically generated

**How did you ensure that your code was efficient?**

I ensured efficiency by not coding things that were unnecessary. One of the biggest examples was using a Map to store Contacts and Tasks. At first, I coded an ArrayList to do this work and used a loop to check each entry for a duplicate ID prior to adding an item. While this did work, it wasn’t perfect and would have become increasingly slow as more items were added to it. To correct this, I swapped to a HashMap which has O(1) lookup time. This also allowed my methods that manipulated this HashMap to be much simpler and easier to understand.

A screen shot of a computer

Description automatically generated

**What were the software testing techniques that you employed for each of the milestones?**

My tests employed black-box tests, specifically equivalence partitioning and boundary value analysis. These two tests worked well with the requirements of the program being built to confirm that the code was meeting specifications.

Equivalence testing is a testing process in which various inputs that should produce the same output are grouped and tested together (Hambling et al., 2019). As an example, the Contact class required a phone number that was not null and was exactly 10 characters long. Since anything less than 10 or anything greater than 10 would fail a test, all inputs of 0-9 characters and 11 or more characters could be grouped and tested as one to prevent testing an infinite number of inputs. In this case 3 characters, 10 characters, and 20 characters were tested with only the 10-character test passing.

In boundary value analysis, the values at the boundaries of valid input are tested since issues tend to happen here (Hambling et al., 2019). As an example, the Appointment class required an appointment ID of 10 or fewer non-null characters. The tests that were performed confirmed that a null ID and an 11-character ID failed while a 10-character ID passed. Here the values around the boundary of the 10-character limit were evaluated.

**What are the other software testing techniques that you did not use for the milestones?**

Other testing techniques that weren’t implemented include decision table testing, state transition testing, and use case testing.

Decision table testing is a technique in which complex decision trees are displayed in a table to help determine what inputs are required to produce an expected outcome. These can then be tested to confirm correct functionality (Hambling et al., 2019). An example could be a program that models the automatic transmission in a car. Depending on the speed of the car a specific gear would be needed. Further if going uphill or accelerating quickly, a lower gear would be needed (the opposite is true for going downhill). These and other factors all play into the optimal gear choice and can be a complex decision. Mapping this in a table can help to make tests that are appropriate for the given inputs.

State transition testing is useful when changes in the states of inputs would produce different outcomes. Similar to decision table testing, state transition testing benefits from the creation of a table to map expected outcomes. In this case, the table is known as a state table and lists the possible events and states along with the associated outcomes from those event/state combinations. Tests can then be designed for these scenarios. (Hambling et al., 2019)

In use case testing, the functions that are performed by various end users are tested for correct functionality for them. It is useful for ensuring that a program will work correctly when used by those for whom it is built. (Hambling et al., 2019)

**For each of the techniques you discussed, explain the practical uses and implications for different software development projects and situations.**

From what I can gather, equivalency and boundary testing are particularly useful when testing small subsets of the program, such as individual methods. Because they focus on very specific inputs (or groups of inputs), they can identify errors in very specific parts of the code.

In contrast, decision table testing, state transition testing, and use case testing shine best when looking at larger subsections of the code as a whole. These tend to focus on more complex functionality (e.g., complex decision trees, use cases, etc.) and can test how complex decisions are made and how end users might use the code when released to them.

I’m sure there are many situations when these distinctions could cross over (e.g., decision table testing for a specific method or boundary testing for a large section of code), but these appear to be generalizations that hold for many cases.

**In acting as a software tester, to what extent did you employ caution? Why was it important to appreciate the complexity and interrelationships of the code you were testing?**

I employed caution by thinking about and testing to the situations that could have been easy to miss. For instance, while testing that tasks could be deleted from the system, I built my test to try deleting more than one task instead of just one. My intention here was to show that deleting a task didn’t cause an issue with the data structure that would have prevented any future tasks from being deleted.

A screenshot of a computer program

Description automatically generated

**Assess the ways you tried to limit bias in your review of the code. On the software developer side, can you imagine that bias would be a concern if you were responsible for testing your own code?**

I tried to limit bias by thinking about my tests prior to coding the program. While I wasn’t entirely certain how I was going to build everything at that point (e.g., what my methods would be named and the order of their arguments), I chose to write brief descriptions of what my tests would do and how I would determine if they passed or failed; similar to pseudocode. This was all replaced with code during the test coding process, but as an example the above deleteTasks() test looked like the following:

Create new task and add to the task service

Delete task by using task ID

Check that deletion succeeded

Normally, bias is a concern when someone is responsible for testing their own code. Because they designed and built their program, they are inherently blind to the issues that might be present which would be tougher to catch while testing. In situations where this is unavoidable, creating tests prior to coding, can help to minimize this bias.

**Evaluate the importance of being disciplined in your commitment to quality as a software engineering professional. Why is it important not to cut corners when it comes to writing or testing code? How do you plan to avoid technical debt as a practitioner in the field?**

Commitment to quality is paramount when creating software. Without this, bugs can be introduced that can cause major issues with the intended use and actions of a program. Cutting corners on testing and miss errors in the code while also giving a false sense of security. Because testing is being done, it could be assumed that issues are being caught, but incomplete testing could glance over issues that would otherwise have been found and corrected. The longer these issues linger, the more costly it can be to find and correct. In the future, I plan to minimize this risk by performing software testing to the best of my ability using a variety of techniques (e.g., boundary testing, equivalence partitioning, decision table testing, etc.)

**References**

Hambling, B., Morgan, P., Samaroo, A., Thompson, G., & Williams, P. (2019). *Software testing an ISTQB-BCS certified tester foundation guide - 4th edition*. BCS Learning & Development Limited.