Kayleigh Kinsey

8/11/23

Software Test Automation and QA

While working on the mobile application features, I consistently made sure that I was meeting all of the project requirements. Before writing any code, I created a list of requirements and what the program would need to do in order to meet those requirements. I then wrote the functional code and returned to my requirements list to determine what tests I would need to ensure every requirement had been met. Evidence of this can be seen in the ordering of my tests, which for the most part line up with the order the requirements were listed. This made it easier for me to step through my tests once they were finished and compare them to the requirements, verifying that every one was covered. Tests have more purpose than just ensuring requirements are met. It is also important that they have adequate code coverage (the percentage of the code that the tests execute). If only the requirements are being tested, lots of potentially devastating bugs can slip through the cracks. Quality testing should reach every corner of a program, aiming to execute most, if not all lines of code at least once. 100% code coverage isn’t always possible in large, complex programs, but since mine were so simple, it wasn’t too difficult to ensure the entire program was tested. My ContactTest, TaskTest, and AppointmentTest classes each have 100% coverage of the Contact, Task, and Appointment classes, respectively. The Service test classes each have 100% coverage of the entire package under test. By aiming for full coverage with my tests, I was able to make sure nothing went untested.

One way that I ensured my code was technically sound was by testing multiple types of invalid input. For example, the id field in the Appointment class could not be more than 10 characters long or null, and I assumed it also couldn’t be an empty string, since an id value is required. I accounted for this with the class’ validateId() method. It ensures the id isn’t longer than 10 characters, and also that it isn’t null or empty. A similar validation method exists for every other field in every class. Whenever a new Contact, Task, or Appointment is created, these validation methods are called to ensure a new instance can’t exist without meeting the class’ requirements for its field values. Continuing with the input validation example, I ensured my code was efficient by writing the validation code only once for each field in each class. The Contact class, for instance, has four validation methods: validateId(), validateName(), validatePhone(), and validateAddress(). There is also a validateAll() method that calls the other validation methods in sequence. Whenever a new Contact instance is created, or one of its mutator methods is called, the input is validated using the validation methods. This means there’s no need to validate this information anywhere else in the code, and no concern that a Contact instance will contain unexpected data (unless at some point user input is fed directly to a Contact instance, because Contact is not made to handle raw user input).

The two main testing techniques I employed for these assignments were input partitioning and boundary analysis. Input partitioning is a technique in which the possible inputs for a function are separated into groups, and each of these groups is tested. Input partitioning can increase the efficiency of tests by reducing their redundancy and the number of tests to be conducted. Boundary analysis works with input partitioning to make tests even more efficient by focusing tests on the boundaries between input partitions. It is at boundaries that errors tend to cluster, so by testing in these areas, we can be sure that the program handles even the most extreme members of each partition appropriately. Both techniques are useful for testing any program that will react differently to different inputs, which is pretty much every program. I used these techniques in writing all of my attribute tests. For example, in my Task class, Task names have a limit of 20 characters and cannot be null or empty strings. I split possible inputs for names into four partitions: 1-20 characters, 21+ characters, 0 characters, and null. In my tests, I set Task instance names using input from each of these partitions, paying particular attention to inputs at the boundaries of each group. These testing techniques helped me to keep my code technically sound by ensuring input is handled exactly the way it is supposed to be, and that invalid input cannot sneak through the cracks.

Some techniques I did not use were decision tables, state transition testing, and use case testing. Decision tables are used to illustrate all the possible combinations of inputs that can be put into an application, and how the application will behave in response. They are useful for modeling systems that involve a lot of conditions or decisions. For example, a program that calculates taxes will require many different variables, and it could become complicated to test every combination of values. A decision table would be helpful in keeping track of all these combinations and ensuring each one gets tested. The classes I created for this project react to input by either accepting it or throwing an exception, which is simple and straightforward enough that a decision table wouldn’t have been too helpful. State transition testing is great for applications that behave differently in different “states”. For example, a car may be “in reverse”, “in drive”, or “parked”. Not all functions work the same between states. The car should be able to turn off when it’s parked, but not any other state, and it should alert the driver to put the car into park before turning it off. State transition testing would be helpful in this situation for ensuring the car behaves properly in each state. However, since my program doesn’t have multiple states, this testing technique wouldn’t have been useful to me. Use case testing focuses on a user’s interactions with an application. For example, a healthcare patient portal offers many functions to users, such as editing account information, messaging doctors, setting up appointments, and accessing test results. Use case testing would be helpful in organizing all these different interactions and planning tests for them. Since my code does not involve any user interaction or input, though, use case testing would not have been helpful in developing my tests.

Because I have trouble with focus, I need to be cautious with all my coding assignments. It is always necessary for me to read back over my work and mentally walk through the logic so that I stay on track. For these assignments, I was particularly careful, because a misconfigured test could make the debugging process really frustrating. I made an effort to scan back over my tests repeatedly for little errors in their names and comments to make sure everything matched. Despite my diligence, there were still a couple of errors that made it through the first versions of my Contact and Task classes, mostly in inconsistent comments. Still, I am happy with the care that I took in writing my tests. I also did my best to limit bias as I worked. I tried to approach testing with a fresh mindset and critical eye, but this turned out to be much harder than I thought I would be. It was difficult sometimes to look for ways to break my own code, and to write tests that I knew could fail. I imagine it would be even harder if I needed to test complex code that I had spent weeks working on. I don’t think bias would prevent me from writing quality tests in the workplace, but it would certainly make it a challenge, and I would have to frequently check in with myself to ensure I’m staying in the “testing” mindset.

Above all, it is important for testers to remain disciplined and vigilant. There are multiple reasons that it can be tempting to test less, usually to save money or time. However, cutting corners is always a risk, and the potential consequences of these risks need to be considered before any rash decisions are made. An airplane system will need extensive tests, because every single thing that goes untested could potentially lead to tragedy. A mobile game, on the contrary, can probably be released with a few minor bugs that can be patched out later, and the cost will not have been too great. Weighing these risks is not just the responsibility of testers, but the whole team. It is the tester’s job to determine how to test a program effectively and efficiently, so that most problems are found in a reasonable amount of time and for a reasonable price. Testers must be committed to the quality of the final product, and personally invested in its ultimate success. Truly, the entire project team needs to understand the importance of testing, because everyone involved is responsible for the quality of their combined output.