CS 320

Project Two

8/12/2022

Jason Veno

Summary:

My testing approach directly related to the software requirements in order to assure the highest possible level of satisfaction pertaining to the completeness of the testing. In this case, the Junit tests were created based on the required functionality of the given units, such as how tasks needed to be added and deleted via their ID as needed. Once it was tested that the units could complete each of these required functions, then testing was complete.

The only foreseeable weakness of my testing was its somewhat low coverage percentage. However, this low percentage was due to the fact that my code contained unused (and technically untested) functions that served as conveniences. For example, the function that removes tasks from the task list is not directly testing within the Junit tests, but its inner functionality was, in this case, the inner functionality being the way in which it searches for and removes tasks from the list. Once it was proven that the function itself works, it is unnecessary to test the function itself, as satisfactory testing was already completed.

Ensuring that code is technically sound can be difficult and complex, but it is much less so when starting at the very beginning of the creation of objects. For example, in Task.java, the constructor has numerous if statements that ensure that each of the associated variables are correctly defined. If any of the variables fail to properly construct, the object will not be created. Each of these variables are individually tested in TaskTest.java to ensure that the if statements in the constructor are working properly.

Ensuring that code is efficient is less rule-oriented than ensuring that it is technically sound because there are many techniques to it, yet it is not always clear when code is efficient enough. A few ways to ensure efficiency is to use simplistic and essential functions only, to use appropriate data structures and algorithms, and to minimize the use of needlessly complicated things like If-Else statements. For example, in Task.java, there is one and only one function for each necessary action regarding the variables, in this case setting and getting them. Each of these functions are well-named and simple, which ensures the maximum possible efficiency.

Reflection:

Each Milestone that was completed in preparation for Project One was essentially a unit, and each was individually tested. For that reason, the Milestones and projects of this course primarily focused on Unit Testing, which is the act of testing units of code before they are integrated together. These Milestones and projects did not include integrating the units, so Integration Testing was not performed. Similarly, System Testing was not performed either. A few structure-based (Whitebox) testing techniques were used, because data structures had to be tested during the Milestones.

Each of the software testing techniques described above have their own uses and applications, but one must be careful when choosing which one to use and at what time. For example, structure-based (Whitebox) techniques are more effective if they follow specification-based (Blackbox) techniques, because Blackbox techniques conduct several preliminary tests on pseudocode, listed requirements, and use cases. After these preliminary tests are complete, Whitebox testing can be conducted with more accuracy, precision, and reliance.

When working on these Milestones and projects, it was vital that I implemented an appropriate level of caution in regards to how the tests were performed. It was imperative that I didn’t accept the code as “working” just because it compiled. For example, just because TaskService.java didn’t crash doesn’t necessarily mean it correctly added tasks to the task list.

In order to prevent bias as much as possible, it was necessary to focus almost solely on the code’s ability to meet the specified requirements. For example, when testing Appointment.java, I had to frequently remind myself to test the code based on the requirements, NOT my understanding of how the code is setup. It is clear how relying on your ability to write clean code can lead to missing an error, and by missing an error, a defect is created, which could lead to total failure.

In order to ensure that professionalism is followed throughout the entire Software Development Life Cycle, a Software Developer must be able to look at his or her code through an outside lens. He or she must recognize the power they hold by being able to create this software, because it can be used for infinitely important things. If the Developer that created the security system at a nuclear power plant cut corners, catastrophe could ensue. Even if I find myself working on a non-life-threatening application like a website, I must still retain that mental fortitude that allows me to create the best possible code in the most professional possible manor.