**7-2 Project Two Submission**

Levi bajuscik

Southern New Hampshire university

Software Test, Automation QA

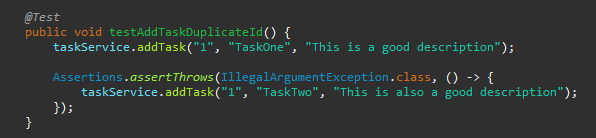
Toni Farley

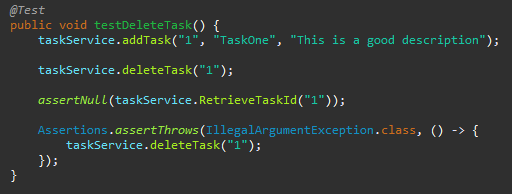
December 15, 2024

**Summary**

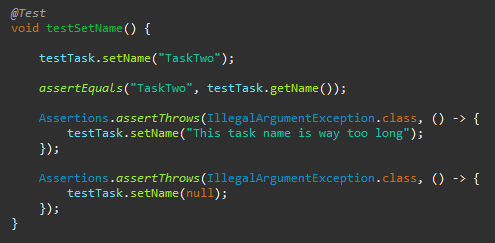
I believe my JUnit tests for both the contact service and task service were aligned pretty closely to the software requirements. For instance, in the task class and the contact class, each of the variables must not exceed their respective character limit and must not be null. Therefore, I made sure to validate this upon creation of an instance as well as when updating the instance when using the various setter methods. I also ensured that each task and contact had a unique ID and could complete further functions like deleting and updating based on that unique ID per the software requirements. I believe the overall quality of my JUnits tests were quite good for both the contact service and the task service. I know this because the coverage percentage for all classes and all methods within those classes was 100%. This means that every test covered all methods in the original class.

I ensured that my code was technically sound by testing and validating each method to see if any of the results would be untrue. For instance, if I tried to add two tasks that had the same task ID or tried to delete a task that didn’t exist.

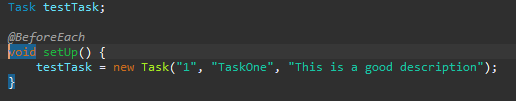




I believe that my tests were efficient because they tested for both the correct usage of a method and also the incorrect usage of the method. For instance, in the testSetName test, I first tested the proper functionality of the setname method by changing the task name to TaskTwo and then used assertEquals to verify if it appropriately changed. I then tested if the changed name is too long or null and throws an exception as can be seen in the provided picture below:



Also, for the TaskTest class, I created a helper method that creates a Task object before each test is run in order to reduce redundancy in my tests because then I don’t need to use the same line of code creating a Task object in every single one of my tests, which helps a lot in efficiency as can be seen in the screenshot below.



**Testing Techniques**

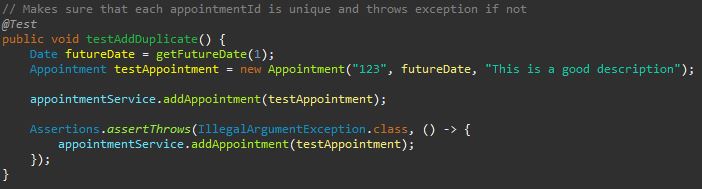
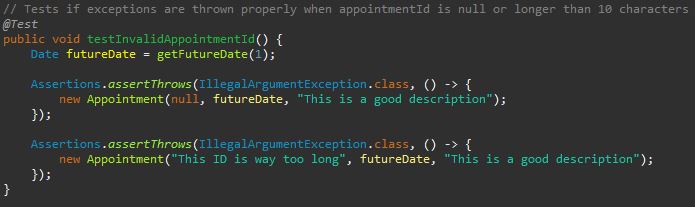
For this project I’ve utilized a software testing technique called unit testing, which is a technique that tests the smallest parts of the application, usually methods to make sure they are working as expected. For instance, a common method throughout all the milestones was to add, delete, or update an object and each of my JUnit tests made sure those worked properly. Another software testing technique I utilized was exception testing, which tests how the software handles exceptions. For each time the software would take an input, I made sure to validate it and properly handle any exceptions that could arise.

Given the overall simplicity of this project, there are quite a lot of software testing techniques that weren’t used. For instance, most, if not all of the non-functional tests were utilized, like security, performance, usability, and compatibility testing. In terms of functional testing, integration, system, and acceptance testing was also not used.

As I mentioned a little bit above, the practical uses for unit testing is that it ensures that both new and changed code functions correctly and doesn’t break pre-existing functionality. Unit testing is also typically done first in functional testing, so you can identify bugs or defects relatively early in the development process although, not as early if the defect was found in static testing. For exception testing, the practical use is that it tests how well the software handles invalid inputs. This ensures that the software is reliable and remains stable when these invalid inputs happen. For a scenario where these two software techniques are used in a much larger development project, utilizing unit testing is arguably even more important because it allows the developer to detect defects before moving on to integration and system testing. Exception testing is much the same regardless of the scope of the project but nevertheless, vital to properly working software. Especially so, if the software handles sensitive data.

The mindset that I’ve adopted for this project and previous assignments from this course is that every piece of code that I write has a chance to mess up and be the cause of a defect. So I made a very conscious effort to test and validate each possible input because even the smallest oversight can have drastic consequences. It’s important to appreciate the complexity and interrelationships of the code in order to help identify any possible bugs, ensure code quality, and improve maintainability. For example, if a function or a class relies on the output of another, it’s crucial to be aware of that dependency and properly test that interaction adequately.

It can be difficult to limit bias on your own and is most effective when you have a second set of eyes reviewing your work. However, in an attempt to limit my bias, at the start of a new coding session, I would review the code I had done the previous day, that way, I would have a fresh pair of eyes to help objectively review my code to see if it was correctly done. Another technique I used to combat confirmation bias was to test for failure. For instance, testing for when an invalid appointment ID is created or when a duplicate appointment ID is trying to be added. If you only test for the correct way, then a lot of defects can happen when the application is used the incorrect way.



When a developer tests their own code, bias is absolutely a concern. We as developers and especially as people are very susceptible to all sorts of biases, which is why it’s important to at the very least be aware of the common biases that arise during the software development process. For example, another very common form of bias is called familiarity bias where, in this case, a developer can be over familiar with the code, which can lead to assumptions that the code works without thoroughly testing each scenario and input.

It is vital that I and other developers do not cut corners when it comes to writing and testing code. If I do not make a concentrated effort to write quality code, then this will lead to many bugs and defects down the line, which will be quite costly in both time and money to fix. The same goes for testing, if I let a defect slip through the cracks due to poor testing, that mistake can become quite critical. To avoid or reduce technical debt, refactoring, pair programming or code reviews are great techniques to utilize. ScrumAlliance defines refactoring as “Refactoring is the reshaping of the code’s structure without changing any of the behavior of the system, so that we can then more easily add the new functionality” (ScrumAlliance, n.d.). Optimizing code by refactoring helps reduce some of the technical debt that may have accrued. Pair programming and code reviews helps ensure that the code meets the quality standards and reduces the likelihood of technical debt being introduced in the first place.

**References**

Mohanani, R. P., Salman, I., Turhan, B., Rodriguez, P., & Ralph, P. (2018, October). *(PDF) cognitive biases in software engineering: A systematic mapping study*. Cognitive Biases in Software Engineering: A Systematic Mapping Study. <https://www.researchgate.net/publication/328410759_Cognitive_Biases_in_Software_Engineering_A_Systematic_Mapping_Study>

ScrumAlliance. (n.d.). *Avoiding technical debt with these “core Four” practices*. Scrum Alliance Resources. https://resources.scrumalliance.org/Article/avoiding-technical-debt-core-four-practices