**Summary and Reflection**

Sarah Spence

xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx

xxxxxxxxxxxxxxxxxxxxxxxxxxx

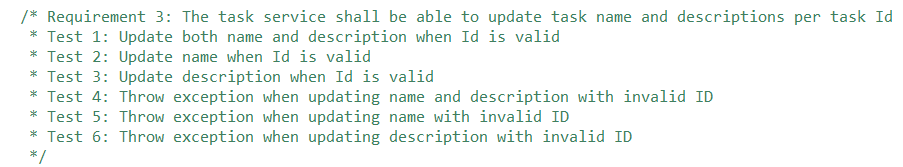
xxxxxxxxxxxxxxxxxx

April 6, 2021

**Summary and Reflection**

**Alignment to Requirements:**

I maintained detailed documentation throughout my project to ensure my testing approach met all of the software requirements. I wrote the software requirements as comments in my test files and listed the number of tests that would correspond to each requirement, as shown below.



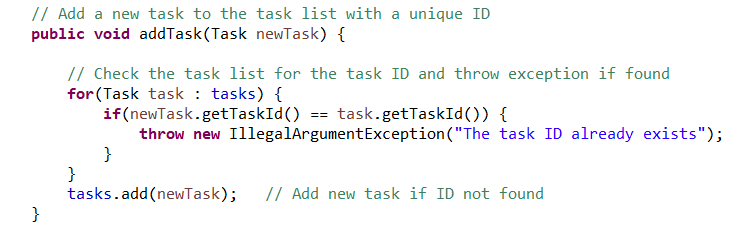
This approach helped me remember the software requirements once I began coding and ensured that my tests covered all edge cases. Regressions testing with unit tests helped me identify my code’s errors sooner than I would have with a manual testing approach. For example, my TaskService class called the update name method in the update description method. The failed test for a description update pointed to this error right away. The unit tests corresponding to each software requirement also ensured that no bugs appeared in my project after updating my earlier features for the final project submission.

**Effective Tests:**

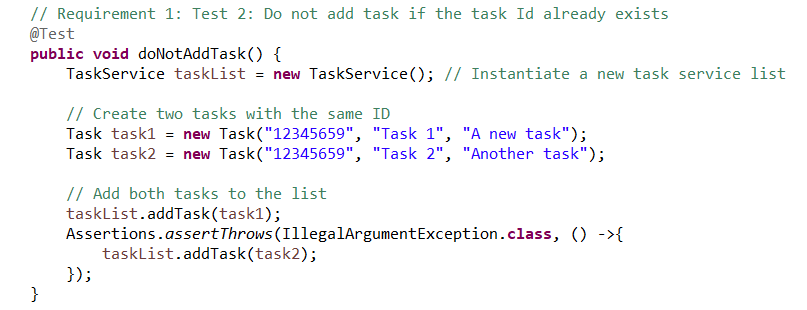
Test coverage refers to the percent of a project’s code tested for errors and defects. Detailed unit tests should obtain 80% to 90% test coverage (Hambling et al., 2015). Eclipse allows developers to check the test coverage percentage by right-clicking on the test packages and selecting “coverage as JUnit tests.” The contact service feature had 90% test coverage, the task service feature had 89% test coverage, and the application service had 91% test coverage. The overall program reached 90% test coverage. Therefore, the test coverage feature was valuable to validate the effectiveness and efficiency of the code. The final project contained 95 JUnit tests that ran successfully. The project had no failed test cases.

**Technically Sound Code:**

I used JUnit tests and exception handling to ensure my code was technically sound. I also made sure to log messages for the user to identify the error when an exception occurred. For example, I used IllegalArgumentExceptions when an argument provided was invalid or null. I also used UnsupportedArgumentExceptions when a user tried to update immutable values or create an appointment for an unavailable time. The code below is an example of exception handling from the TaskService class. This code throws an exception when adding a task to the list that does not have a unique Id.



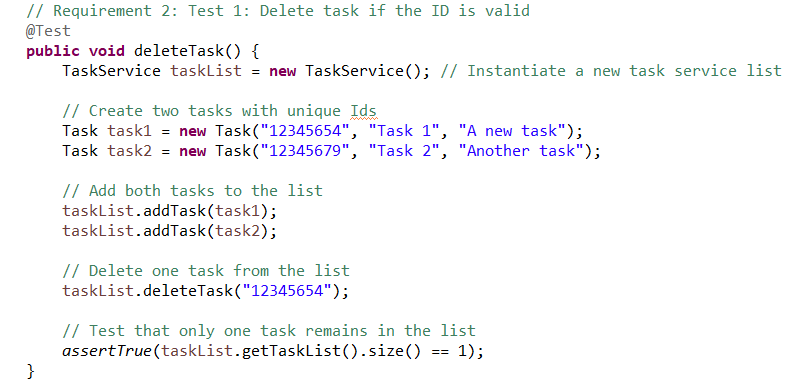
Next, a JUnit test confirms that the method throws an exception when the Id is not unique using a lambda expression.

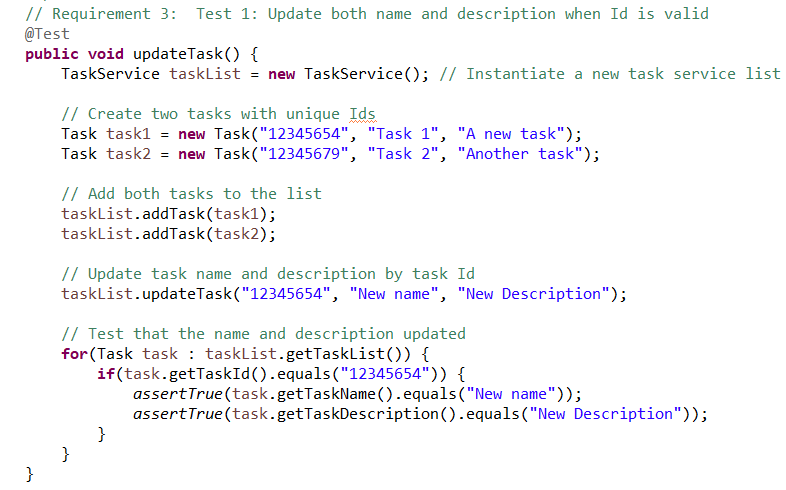


Exceptions ensure code is technically sound by informing both the user and the compiler that an exception has occurred to stop code execution. Assertions to check for exceptions were valuable and verified that the code did not fail tests for unusual values.

**Efficient Code:**

The unit tests show how long it took for each test to run. This feature helped to ensure the code was efficient. I used the same size list to test the delete, add, and update methods for each of my features. For example, the methods in the TestTaskService class instantiated a task list. Then, these methods created two new tasks and added them to the task list. The code below illustrates these steps.





Testing with the same size list could indicate an issue with efficiency if one test took longer to run than others since the delete and update methods have the same runtime of O(n). I used a similar approach when testing the contact service and appoint service classes.

**Techniques Employed:**

I utilized several black box software testing techniques throughout the project milestones. I used equivalence partitioning to set the valid and invalid appointment dates. Next, I applied boundary value analysis when testing my appointment service class to ensure the program would not schedule an appointment for the same day if the hour had passed. A boundary test helps determine if values just outside or inside of the given range cause a test failure (Hambling et al., 2015). I instantiated two new calendars and subtracted one hour from the second calendar, so this test will always test one hour behind on the current day. Hambling states, “for most practical purposes, the boundary value analysis must identify two values at each boundary” (Hambling et al., 2015). Boundary value analysis complements equivalence partitioning.

Another black box technique used in the project milestones was regression testing. Regression testing was the most valuable technique used in these assignments because it allowed me to retest previously successful tests after making changes to the files. Regression testing pinpointed some new errors that I would not have caught without rerunning the tests after each update. I also used use case testing, another black box method that involves writing tests to correspond to each use case. I typed each use case in my test files first and brainstormed tests for the requirements. This approach guided the format of my code and ensured I did not miss project requirements.

A white box testing technique applied to the course project was decision testing and coverage. White box testing techniques “explore system or component structures at several levels” (Hambling et al., 2015). Decision testing “aims to ensure that the decisions in a program are adequately exercised” (Hambling et al., 2015). I applied decision testing to test the if/else branches that handled exceptions for invalid and null input values. I also used decision testing for if/else branches that managed non-unique ids and unavailable appointment dates.

**Other Techniques:**

There are several more testing techniques not applied in these milestone assignments. I did not apply state diagrams, a black box technique used to identify if different inputs change a program’s output (Hambling et al., 2015). The boundary and decision testing already tested different outputs in my unit tests. This system did not seem complex enough to make the time spent designing a state transition diagram worthwhile. Similarly, I chose to omit decision table testing because my other testing techniques covered the system requirements without the need for tables and graphs. Decision table testing is another black box technique in which developers create decision tables to “list all the input conditions that can occur and all the actions that can arise from them” (Hambling et al., 2015). This technique may have been overkill for this project because it only had to test for invalid inputs and did not have enough decision branches to gain value from a decision table.

I also omitted several white box testing techniques. Hambling states, “A control flow graph provides a method of representing the decision points and the flow of control within a piece of code, so it is just like a flow chart except that it only shows decisions” (Hambling et al., 2015). Control flow graphs would have been valuable if my program incorporated decision branches with three or more decisions. Similarly, hybrid flow graphs also test a program’s decision branching and describe the branches’ logical flow. This testing technique would have also been valuable for programs with more decision branches.

**Uses and Implications of Techniques**

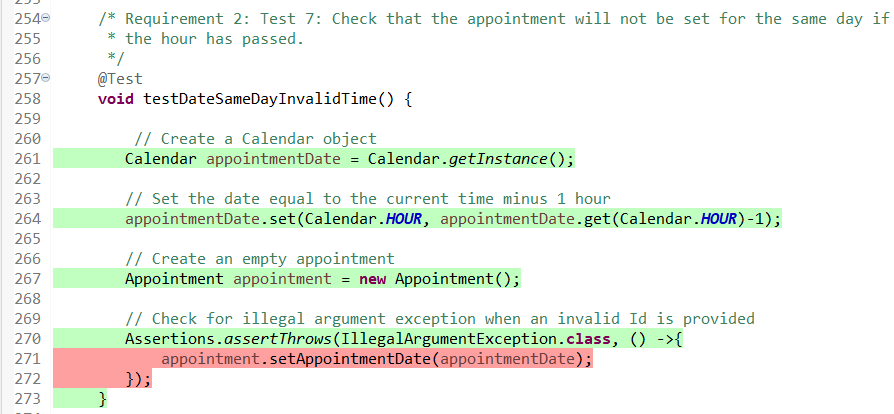
Equivalence partitioning and decision boundary analysis are excellent for projects that have restrictions on input values. The appointment, contact, and task services that required restrictions on input lengths are examples where these techniques provide value. Equivalence partitioning and decision boundary analysis would not provide value to projects without numeric input restrictions. Regression testing is beneficial in all cases where a project is still in progress and changes to files occur. The only scenario I can think of where regression testing would not benefit a project is for small automation scripts that will not change or apply to larger projects.

Decision testing and coverage are helpful in programs where the output depends on the given input. A program that incorporates if/else branching would utilize decision testing. Decision testing will not offer benefits if the program does not change the output based on input values or include decision branching. A program to set a student’s name may not require decision testing if the name length is not restricted and can be null. Similarly, smaller scripts to update file names or validate input may not require use case testing because they have only one use case.

I also omitted several diagramming techniques from this project because the project requirements and flow were straightforward. State transition diagrams “are useful in systems where combinations of input conditions produce various outputs” (Hambling et al., 2015). A state transition diagram might have been useful if my program incorporated more decision branching or a wider variety of output options. Decision table testing, control flow graphs, and hybrid flow graphs would also have been valuable for a project that incorporated more output options. A program that will set interest rates depending on a candidate’s credit score has a large volume of outputs available and may benefit from state transition diagrams, decision table testing, or flow graphs. Projects that do not incorporate more than one or two decision branches would not benefit from these diagrams.

**Caution:**

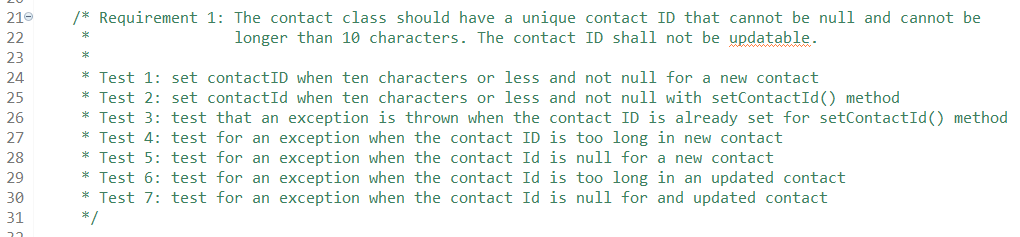
I adopted the mindset of a software tester when working on this project. I exercised caution by trying to find ways of making the code fail its test cases. For example, I coded a test in the appointment service class to test that the program would not schedule an appointment for the same day if the hour had passed. The image below shows the structure of this test case.



The test case returned the current date and set the appointment date for the current date minus one hour. Next, the test case checked for an exception when setting the appointment with an invalid time. Applying regression testing by rerunning all tests after making changes was another precaution that guided this project’s development. The tests and their naming conventions pointed to problem methods right away and saved me debugging time.

**Bias:**

Hambling explained that developers try to prove there are no errors in their code, and testers try to prove that errors exist in the code (Hambling et al., 2015). This statement describes testing bias because developers sometimes miss test cases because they believe their code is functional and will pass its tests. I tried to limit bias in my tests by writing the code requirements and corresponding tests as comments in my test files before beginning any code, as described above. The image below shows another example of this documentation.



Brainstorming the tests before I wrote the code helped reduce bias in my tests. I used an incremental approach to test development by adding additional required test cases inspired by the development process. For example, I coded overloaded constructors in my class files to allow class instantiation with or without parameters. This feature allows other developers to instantiate the classes in two ways. However, overloaded constructors required adding some more test cases to verify both instantiation methods.

**Discipline:**

The article “When Coding Goes Wrong” by Coder Academy explains some examples of why discipline is critical for software engineering professionals. Exercising discipline means developing thorough test cases that meet code coverage requirements and prevent bugs in the software’s release. Being disciplined also means not cutting corners with software tests and limiting assumptions that the code works. The article describes some scenarios where bugs made it through to the release phase and caused the companies and developers problems. For example, a software glitch caused Japan’s Hitomi satellite to fire thrusters in the wrong direction, causing the satellite to spin out of control and costing Japan 268 million dollars (Coder Academy, 2016). The software developers could have prevented this massive expense by testing the code better before deploying the satellite. I plan to eliminate technical debt in the field by testing my code in depth before pushing it to a remote repository and trying to eliminate errors in earlier phases of development when they are still relatively cheap to correct.

**References**

Coder Academy. (May 23, 2016). When Coding Goes Wrong. *Coder Academy.* <https://medium.com/@coderacademy/when-coding-goes-wrong-e46d84c6565f>

Hambling, B., Morgan, P., Samaroo, A., Thompson, G., & Williams, P. (2015). Software Testing - An ISTQB-BCS Certified Tester Foundation Guide (3rd Edition). BCS The Chartered Institute for IT.   
<https://app.knovel.com/hotlink/toc/id:kpSTAIST01/software-testing-an-istqb/software-testing-an-istqb>