CS-320-J7683 Software Test Automation& QA 23EW4

7-2 Project Two Submission

Jacob Simmons, 16 Apr 2023

**Summary: Alignment to Requirements**

The below is an example (citing a specific example) where the code testing is aligned with the requirement.

|  |  |
| --- | --- |
| Requirement | Task Class: The task object shall have a required unique task ID String that cannot be longer than 10 characters. The task ID shall not be null and shall not be updatable. |
| Code Deliverable | @Test  @DisplayName("Task ID cannot have more than 10 characters")  void testTaskIDWithMoreThanTenCharacters() {  Task task = new Task("Name", "Description");  if (task.getTaskID().length() > 10) {  fail("Task ID has more than 10 characters.");  }  }  @Test  @DisplayName("Task Name shall not be null")  void testTaskNameNotNull() {  Task task = new Task(null, "Description");  assertNotNull(task.getTaskName(), "Task Name was null."); |

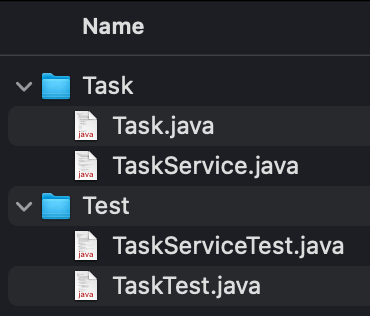
**Summary: Effective Tests**

The code was effective based on the requirements as the testing was in alignment with the requirements. Additional measures could be used to broaden the aspect of the testing to ensure all areas of the requirements are covered, but the minimum requirements were considered for the testing panel.

These tests were coded based on the requirements. There are edge cases that were not considered (security breaches with input validation, etc.) that could be explored, however the tests meet the direct requirements.

**Summary: Technically Sound Code**

An example of the code and structure being technically sound is the file construction for the project. I split the code into two different packages, one for the tasks and then another for the tests related to these tasks. I focused on using industry standards while trying to keep it easy to read.



**Summary: Efficient Code**

Looking throughout the code, there are no instances where the time complexity is a concern (all is O(1)). The idea behind the code was to ensure the testing parameters were understood and then implemented in a way as to ensure the adequacy of the output to the requirements. Good industry practices were used throughout the code, and example of this would the section of code where the packages and libraries were imported before the class definition:

package Test;

import static org.junit.jupiter.api.Assertions.\*;

import org.junit.jupiter.api.DisplayName;

import org.junit.jupiter.api.Test;

import Task.Task;

class TaskTest { …

**Reflection: Techniques Employed**

The approach was to take each requirement line and then generate a test case for the needed deliverable; each deliverable was taken and then individually tested to understand where the fault might be (having them both together could yield a result where one does not know exactly what failed and what passed thus both were separated).

A white box approach was used throughout this testing process. According to (CSRC Content Editor, n.d.-b), " [white box testing] also known as clear box testing, glass box testing, transparent box testing, and structural testing) is a method of testing software that tests internal structures or workings of an application, as opposed to its functionality (i.e. black-box testing)." This was the governing structure for the testing structure layout.

**Reflection: Other Techniques**

Throughout the testing method approach the block box testing methodologies were not directly employed as the functionality of the code is known. The inputs, processes, and outputs are directly known and therefore the white-box testing methodologies were tasked as higher priority when looking at the application methodologies.

**Reflection: Uses and Implications of Techniques**

When comparing white box and black box testing methodologies, black box testing methodologies are employed when the internal workings of the code are not known and only the inputs and outputs are divulged. Test cases are developed based on this scenario. As the internal workings of the code are known, the white box testing methodologies are employed throughout this testing approach.

**Reflection: Caution**

Caution was given when it can to the tests that were deployed. As the requirements for the functionality of the program were known, it was determined the most effective way to generate the tests were through these pre-determined channels. Going outside of these testing limitations could be considered "not client" specific and a failing result may not necessarily cause a rejection by the client/user. Caution was given when it can to the development of these tests as the tests need to be in alignment with the client/user expectations. The exact requirements were directly tested.

**Reflection: Bias**

When coding, the developer needs to be aware of the potential bias in the coding. The coding should be assessed for use based and linked back to the client/user requirements. Ideally a traceability matrix should be established with a cross-functional team where the needs of the client/user are clearly understood and the bias is reduced through the use of the cross-functional team. When comparing to this project the bias was reduced as the requirements for testing were already defined.

**Reflection: Discipline**

As a Developer or Tester is creating the code and the testing, the work can become tedious and its important that the testing is laid out in a manner that is aligned with the requirements and that the file organization of the file structures is in alignment with the organizational needs/policies. The reason for this is that is allows the assimilation of this code into the overall system structure for the organization is critical. This will also all future review of this code to be more streamlined.

**References**

CSRC Content Editor. (n.d.-b). White Box Testing - Glossary | CSRC. Retrieved April 2, 2023, from https://csrc.nist.gov/glossary/term/white\_box\_testing