Andres Trujillo

9/29/2024

Southern New Hampshire University

CS320 Software Testing, Automation, and QA

Prof. Parul Hirpara

Module Four Journal

The testing approach used for the Task object class and the Task Service class aligned to the specified requirements. Aligning to the requirements involved creating a list of items that were related to the project’s requirements. So, for instance, the Task object is to have a field member variable for the Task name. The requirements were that the name cannot be null or be longer than twenty characters. This means that within my code I had to validate the way I created a Task object.

Task validTask = new Task("validId", "name", "description");

Assertions.*assertNotNull*(validTask);

Within this example my test, creates a valid Task and ensures it was generated successfully. This met all the requirements for creating a task object from the customer. The name/Id/description could not be null so that was a conditional line of code, and I had another conditional line of code checking the length of the name/Id/description for each respective field member that looked like this within the Java application. This is one example of the ways my code met the requirements.

if (Id == null || Id.length() > 10) {

throw new IllegalArgumentException("ID is invalid. ");

}

Once these requirements were satisfied, I could let my program continue running towards the creation of a Task object. Otherwise, an illegal argument exception object was created by the Task. A task consists of variables implemented by understanding our requirements. These variables had certain requirements given to us by our client of the mobile application. The Id had to be unique, so we needed to check for uniqueness amongst our database to identify any duplicates when our Task Service object was adding tasks to its database. We accomplished this with this code. This was the structure of our conditions that allowed us to test the given requirements.

if (!task\_map.containsKey(id)) {

task\_map.put(id, task);

} else {

throw new IllegalArgumentException();

}

The Task Service class was bound to certain requirements that were the same as our Task class. Although, we needed the ability to add, delete, and update Tasks. But the creation of Task Services tests still had to meet our requirements, intertwining the two classes making it more complex. The Task Service required a database implementation, and that was achieved with a hash map pointing a unique Id to a Task object keeping track of all our Tasks scheduled within the Task Service. We assessed this the same way in generating a Task, we made sure that our created object with a valid constructor was not null. When adding tasks, we had to ensure that there were no duplicates, and the task was not null. This required a valid Task object and a valid unique Id otherwise we would generate an error and adding the task object to our database would not succeed because it had the same structure as our previous conditionals. The errant code not aligned to the customer requirements would throw an illegal argument exception.

My test coverage percentage was one hundred percent for my Task class, covering all constructors and methods. The test coverage helps me understand the amount of executing machine instruction, and I can write tests that follow conditional branches, or check’s function variables. This test coverage was comprehensive to me because we had to check if a variable for each class was not null. Then we had to create a variable that was not the correct length and ensure that our conditional branch throwing an exception was working as expected. The use of logic and appropriate testing methods allowed me to ensure that the customer requirements were the most important thing to test.

Technically sound code can be difficult to prove because there are many ways to do things. So, I will define technical sound code as code that is well commented, tested, and easy to read by using appropriate variable names and appropriate function names. Using these techniques alongside the help of a testing class. I made sure my function name to update a name was, “updateTaskName” helping others reading the code to understand immediately what the function does to the program. Using the static analyzer tool was important to ensuring that technical sound code was used throughout the program. The tool made sure that functions were covered in tests, or variables in the test classes. This bit of test code ensured that we can delete a task.

task\_service.addTasks(task.getId(), task);

Assertions.*assertDoesNotThrow*(() ->{

task\_service.deleteTasks(task.getId());

});

Assertions.*assertThrows*(IllegalArgumentException.class, () ->{

task\_service.deleteTasks("1234567890");

});

To simplify the ability to test whether our task service can properly delete a task. We have previously created an object and this code starts with adding the task. And asserting that it doesn’t throw an error, because it is receiving a valid task. Then we call the “deleteTasks” function with the valid task Id. This method passes our tests so it continues to the next assertion. Where we attempt to delete the valid task by unique Id that each test is using for their tests. This encounters the illegal argument exception, therefore ensuring the test passes. Re-using valid tasks by having a “BeforeEach” annotation ensures that we have an efficient setup. Thereby not having to create a valid task for each test but Junit will call the “BeforeEach” function where I set up the valid task. For completeness, I also included a test for the task service constructor.

@BeforeEach

void createTaskService() {

task\_service = new TaskService();

task = new Task("1234567890", "Task Name", "Description of task.");

Assertions.*assertNotNull*(task\_service);

Assertions.*assertNotNull*(task);

}