**Project Two**

Steven Klabunde

Southern New Hampshire University

Software Test, Automation QA

Eidan Rosado

August 14, 2025

To write a successful program, it is important to review all of the requirements that are presented to you. When I was writing the code for the module three milestone, I started by looking at what variables would be needed and what specific requirements they had. The first thing I found was that in the contact class variables, they all shared the requirement of not being null. Then they all had a maximum length of 10 except for address which could have up to 30 characters. Inside the code I wrote for each of these, I included conditions where if nothing was entered into the input for a variable, it was given a value of NULL. If an input had more than the number of characters allowed, it only accepted the first 10 characters or in the case of address 30 characters. This is shown in the following code.

  //Create conditions for First Name

    if (firstName == null || firstName.isEmpty()) {

      this.firstName = "NULL";

    }else if (firstName.length() > 10) {

      this.firstName = firstName.substring(0,10);

    }else {

      this.firstName = firstName;

    }

Writing code this way is important because you address what the stakeholder is looking for right from the beginning which aids in the testing process.

When I began my junit testing in module four, I started with task and then moved onto task service. I started from the top of the code and worked my way down. I looked to see if I felt any tests were appropriate for each line of code. In this way, I felt that I was thorough and did not miss anything. I believe that gave me a full coverage percentage of testing done. I had proceeded with junit testing the same way for module three.

To make sure that the test is technically sound, I built my junit testing appropriately by going into the correct file and creating a junit test case. This made sure that I was starting the testing correctly so that it would work properly. Again, I went line by line and determined what tests were needed. For module four, the conditions of the task ID were that it could be no longer than 10 characters and could not be null. It also states that it is not updatable, so no method was created for updating the task ID. Tests were created to verify that the ID was not null and was not too long. Here is an example of that in code.

@Test

**void** testTaskIdNull() {

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

**new** task(**null**, "test task name", "test task description");

});

}

@Test

**void** testTaskIdTooLong() {

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

**new** task("123456789", "test too long task name", "test task description");

});

}

This makes sure that all requirements are met. This code is technically sound as it presents the test that is being performed and then puts forth the assertions. If the assertions are not met, the test will fail. The test will pass only if all assertions pass.

The testing is efficient as by going through each line and creating tests, no unnecessary tests were created. For example, each method was tested to verify accuracy. One example of this in code is updating a task name.

@Test

**void** testUpdateTaskName() {

newTaskService.addTask("123456789", "test task name", "test task description");

newTaskService.updateTaskName("123456789", "new task name");

**for**(**int** i = 0; i < newTaskService.taskList.size(); i++) {

**if** (newTaskService.taskList.get(i).getTaskId() == "123456789") {

*assertTrue*(newTaskService.taskList.get(i).getTaskName().equals("new task name"));

}

}

}

@Test

**void** testUpdateTaskNameNull() {

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

newTaskService.addTask("123456789", "test task name", "test task description");

newTaskService.updateTaskName("123456789", **null**);

});

}

@Test

**void** testUpdateTaskNameTooLong() {

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

newTaskService.addTask("123456789", "test task name", "test task description");

newTaskService.updateTaskName("123456789", "this task name id too long");

});

}

Tests were created to verify that the update was correct, that the new task name is not null and that the new task name did not exceed the number of characters. All of these were necessary per the variable requirements, and no extra tests were done for this method. This is how I ensured efficiency. Everything should be done to meet stakeholder’s requirements. It is important that this is always kept in mind both while writing the code and also while writing the tests.

There are many software testing techniques that can be used when writing code. The most noticeable technique that was used in these modules was the junit testing. This is a testing program that can be used to test different aspects of the program. Junit is an automation testing tool. This means that once the code is written, it will continue to automatically run those tests. One technique that I employed was boundary value testing. In the requirements each variable had a maximum number of characters that should be allowed. Here is an example of that test for contact service for the taskID variable.

A close-up of a text

AI-generated content may be incorrect.

It was also stated that the variables were not to be null. Here is an example of that for the same variable.

A close-up of a computer code

AI-generated content may be incorrect.

Between these two tests boundaries are set for the variable.

Each of these tests is white box testing. We are looking at the code and then creating a test for it. Most of these tests are functional as they are looking to make sure that the stakeholders’ requirements are met. Boundary value testing is extremely important because it prevents overloading the system. A user could make an infinite loop so that others were not able to access it. It also protects us from a malicious attack changing the code to allow someone admin access. What we have done with these is unit testing. We created classes inside of packages and then created a junit test for each class.

One true test that has not been done is integration testing. While I have made sure that each class has been tested, at this point, no testing has been done for whether each of the classes will integrate well with each other. The other type of test that has not been done is regression testing. Regression testing makes sure that the code still works even if something within the code has been changed. I have not written these types of tests but look forward to learning mor about them in the future.

The mindset I took when I was writing the code was systematic. I tried to look at what was required and to create each piece of code meeting those requirements. I wanted to use methods as much as possible to avoid repetition. Then when writing the tests, I tried to continue the systematic method so that each test was correlated with a section of code. As I was writing the code, I tried to fix any errors as they came up. With that, I do believe that I used caution to make sure that all requirements were met and that all code was appropriately tested and functioning. One thing that I was aware of was the way that the code was working together. What seems to be a simple task of deleting a contact involves creating a method but then creating testing for that method as well.



A computer code with text

AI-generated content may be incorrect.

It is important to make sure that all of the relationships between items are considered and verify that one method does not cause problems with a different one.

Bias can become a problem if the developer does not develop proper tests due to being the one to write the code. This can stem from misreading or not understanding the requirements that were presented by the stakeholder. For instance, if the developer missed the requirement for task service that the description field must be updatable, not only the code for this will not be written but also the test will never be written. That is the danger of the developer also being the tester. In order to prevent this, I started with reading the requirements thoroughly, wrote the code with the requirements in mind, and then went back to the requirements once the code was written to verify everything was accounted for.

There are so many consequences that can come from improper or incomplete writing and testing of code. It can be more minor, such as software running slowly or a small piece of a webpage not functioning properly to something more major such as customers’ confidential information being compromised. It can even lead to loss of life in some cases such as software issues in a new vehicle that causes brakes to fail or airbags not to deploy. In order to avoid this, I will make sure that I fully understand the needs of the software and that it is tested properly prior to release. Technical debt can be incurred if these processes are not followed. I will not cut corners and will be diligent in my work and testing.