Project Two

John Dickerson II

Southern New Hampshire University

CS-320-11465-M01

Professor Linda Gaston

August 18, 2024

I believe all the sections for my project were aligned with the listed requirements. All the sections had a length and ‘non-null’ requirement for each attribute. In the ‘Task’ class, the task ID had to be ten characters or less, and it could not be null. In the ‘Task” class I had an if statement to check these two requirements:

if(taskId == null || taskId.length()>10)

If the requirement in the if statement were met, an IllegalArgumentException was thrown:

throw new IllegalArgumentException("Invalid Task ID");

In the related Junit test ‘TaskTest’, I implemented two test cases for task ID. One for the ID being too long, and one for the ID being null:

//ID tests

@Test

void testIdTooLong() {

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

new Task("12345678910", "Name", "description");

});

}

@Test

void testIdNull() {

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

new Task(null, "Name", "description");

});

}

These if checks into Junit test format was used for every attribute for each of the classes in the application. All of these tests have a “passing grade” for coverage. The lowest coverage being 80.0% for TaskService and the highest being 100% for AppointmentService. The average coverage for every class is 88.08%. In the test coverage section of the project one rubric “Ensures that the test coverage for the JAVA files has 80% coverage or higher” is the requirement for full credit for that section. Based on those facts, I believe my tests meet the standards of “effective”.

The main two software testing methods I employed were manual testing and unit testing. As I stated earlier in this report, the coverage for my Junit tests was above 80%. So that covers the unit testing for each of the classes. I also ran the code with an added main function. In that function I added tasks or appointments both inside and outside of the class requirements to ensure the code would add only the approved items and throw exception errors only for those items outside of the restrictions. This testing method is useful to ensure common user errors will be handled appropriately, and to ensure the exceptions are not being thrown for inputs that are correct. One testing technique I failed to use was system testing. Simply put, this would be like a unit test for the application as a whole.

Prior to this class I only used manual testing when creating programs. For smaller and less complex programs this is a great enough testing method. The larger more complex an application gets, the more manually testing gets increasingly harder. Especially when multiple people/teams are working on different parts of the same project. For example if one person coded the “ContactService” class and another person coded the “ContactServiceTest” class, if ContactService did not check to see if the contactId was in the contacts hash map, the user could call the deleetContact method at inappropriate times.

public void deleteContact(String contactId) {

// throws error if contact is not in hash map

if (!contacts.containsKey(contactId)) {

throw new IllegalArgumentException("Contact does not exist");

}

contacts.remove(contactId);

}

This exception in ContactServiceTest would never be called:

@Test

public void testDeleteContact () {

ContactService contactService = new ContactService();

Contact contact = new Contact("0123456788", "John", "Doe", "1234567890", "123 Main St");

//contactService.addContact(contact);

assertTrue(contact.getId().equals("0123456788"));

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

contactService.deleteContact("0123456788");

});

}

I spent many hours developing this application. Of course, I think it is the best and deserves a 100%, but if I put my ego to the side, I can let the numbers speak for themselves. The numbers being the coverage percentages. With an average coverage of 88.08%, if the entire project received that grade, I would be happy. It would be easy to do a bit of manual testing and think your code is bullet proof. But there are things that could happen (in the code or with user input) that the developer may not think of. It is important to limit bias and implement impartial testing.

Achieving the requirements should be seen as the minimum for applications such as this. There may be common issues that are not listed in the requirements that need to be addressed. Or there may be a feature that I realize may be more useful that the customer may not have thought of. For the appointment class, you can search for the appointment by ID, but what about by date? Maybe the office needs to be closed on a certain day. The client would need to know every person who has an appointment that day so they can contact them to reschedule.