Craig Rowell

2022/09/23

CS-320 Software Test Automation & QA

Module Four Journal

When I first submitted Contact Service, my testing approach was not very well aligned to the software requirements. I have not been writing much object-oriented code for the past 6 months, and I hadn’t written a line of code in Java for over 9 months. I remembered the concepts I had learned, and building a constructor wasn’t too difficult, but implementing the getters and setters properly and building a hash map proved to be very challenging. Luckily, I have written quite a few unit tests so I didn’t have any issues structuring the unit tests. On the testing end, I ran into problems trying to test object properties that were stored behind private and protected access modifiers because I had implemented the get contact function improperly so my hash map/ list array wouldn’t produce objects to test. On Sunday night, I believe I had maybe 60% coverage at best. I kept at it Monday and resubmitted with 80%ish code coverage, I believe. I pulled an all-nighter on Tuesday and got it up to 91%, but my code base had a couple errors I couldn’t suss out and my work was already graded so I didn’t resubmit. Most of my struggle with unit testing stems from testing scenarios with overlapping factors, but I believe that is something that will come with time and experience.

Task Service was much simpler because the requirements were identical to Contact Service, so it was simply a matter of changing class/module names and deleting/renaming some variables/parameters. 100% coverage, no errors, good encapsulation, lots of abstraction. It could use some better inheritance, a parent class to consolidate the similar functionality between the two services, but I would backlog in an action item, and discuss at the next stand up meeting where the team discusses technical debt.

I ensured that my model was successful by verifying that each property in the model updated, save for the ID:

@Test

void testUpdateTaskSuccess() {

Task task = new Task("123456", "Bob", "Test successful task update");

task.setName("Jeff");

task.setDescription("Test task updates");

assertTrue(task.getTaskId().equals("123456"));

assertTrue(task.getName().equals("Jeff"));

assertTrue(task.getDescription().equals("Test task updates"));

}

I also created asserts for all exceptions, one example being a unique ID with too many characters:

@Test

void testCreateTaskTaskIdFails() {

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

new Task("12345678901", "Bob", "Test task creation failure due to taskId length");

});

}

I knew my service code was technically sound when I was able to create, access and update multiple contacts/ tasks by their unique identifiers, but only when the identifiers were unique:

@Test

void testAddMultipleTasksSuccess() {

Task task1 = new Task("123458", "Bob", "Mike check 1 2");

Task task2 = new Task("123459", "Sue", "Mike check 1 2");

assertTrue(taskService.addTask(task1));

assertTrue(taskService.addTask(task2));

}

@Test

void testAddTaskDuplicateIdFail() {

Task task1 = new Task("123457", "Bob", "Mike check 1 2");

Task task2 = new Task("123457", "Sue", "Mike check 1 2");

assertTrue(taskService.addTask(task1));

assertFalse(taskService.addTask(task2));

}

@Test

void testGetTaskAndUpdateSuccess() {

Task task1 = new Task("1234", "Bob", "Mike check 1 2");

assertTrue(taskService.addTask(task1));

Task updateTask = taskService.getTask(task1.getTaskId());

updateTask.setName("Sue");

updateTask = taskService.getTask(updateTask.getTaskId());

assertTrue(updateTask.getName().equals("Sue"));

Craig Rowell

2022/09/23

CS-320 Software Test Automation & QA

Module Four Journal

When I first submitted Contact Service, my testing approach was not very well aligned to the software requirements. I have not been writing much object-oriented code for the past 6 months, and I hadn’t written a line of code in Java for over 9 months. I remembered the concepts I had learned, and building a constructor wasn’t too difficult, but implementing the getters and setters properly and building a hash map proved to be very challenging. Luckily, I have written quite a few unit tests so I didn’t have any issues structuring the unit tests. On the testing end, I ran into problems trying to test object properties that were stored behind private and protected access modifiers because I had implemented the get contact function improperly so my hash map/ list array wouldn’t produce objects to test. On Sunday night, I believe I had maybe 60% coverage at best. I kept at it Monday and resubmitted with 80%ish code coverage, I believe. I pulled an all-nighter on Tuesday and got it up to 91%, but my code base had a couple errors I couldn’t suss out and my work was already graded so I didn’t resubmit. Most of my struggle with unit testing stems from testing scenarios with overlapping factors, but I believe that is something that will come with time and experience.

Task Service was much simpler because the requirements were identical to Contact Service, so it was simply a matter of changing class/module names and deleting/renaming some variables/parameters. 100% coverage, no errors, good encapsulation, lots of abstraction. It could use some better inheritance, a parent class to consolidate the similar functionality between the two services, but I would backlog in an action item, and discuss at the next stand up meeting where the team discusses technical debt.

I ensured that my model was successful by verifying that each property in the model updated, save for the ID:

@Test

void testUpdateTaskSuccess() {

Task task = new Task("123456", "Bob", "Test successful task update");

task.setName("Jeff");

task.setDescription("Test task updates");

assertTrue(task.getTaskId().equals("123456"));

assertTrue(task.getName().equals("Jeff"));

assertTrue(task.getDescription().equals("Test task updates"));

}

I also created asserts for all exceptions, one example being a unique ID with too many characters:

@Test

void testCreateTaskTaskIdFails() {

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

new Task("12345678901", "Bob", "Test task creation failure due to taskId length");

});

}

I knew my service code was technically sound when I was able to create, access and update multiple contacts/ tasks by their unique identifiers, but only when the identifiers were unique:

@Test

void testAddMultipleTasksSuccess() {

Task task1 = new Task("123458", "Bob", "Mike check 1 2");

Task task2 = new Task("123459", "Sue", "Mike check 1 2");

assertTrue(taskService.addTask(task1));

assertTrue(taskService.addTask(task2));

}

@Test

void testAddTaskDuplicateIdFail() {

Task task1 = new Task("123457", "Bob", "Mike check 1 2");

Task task2 = new Task("123457", "Sue", "Mike check 1 2");

assertTrue(taskService.addTask(task1));

assertFalse(taskService.addTask(task2));

}

@Test

void testGetTaskAndUpdateSuccess() {

Task task1 = new Task("1234", "Bob", "Mike check 1 2");

assertTrue(taskService.addTask(task1));

Task updateTask = taskService.getTask(task1.getTaskId());

updateTask.setName("Sue");

updateTask = taskService.getTask(updateTask.getTaskId());

assertTrue(updateTask.getName().equals("Sue"));

I ensured my code was efficient by using a hash map data structure to store the IDs as keys and other properties as values, and by reviewing my work once it was complete, to remove redundancies:

assertTrue(getTask.getTaskId().contentEquals(task.getTaskId()));

assertTrue(getTask.getName().contentEquals(task.getName()));

assertTrue(getTask.getDescription().contentEquals(task.getDescription()));