Jason Gionfriddo

CS-320 Software Test Automation & QA

Dr. Cross

Octoberber 17, 2021

The tests that I wrote focused on ensuring that the base cases for the requirements were covered and up to the requirement standards. For example, the task object had the requirements of not being longer than 20 characters and not being null. My test ensured that a null name would throw an exception with the following code:

*@Test*

void testNullName() {

//Testing null input for name.

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

new Task(“1234567890”, null, “Pick up eggs and milk”);

});

}

I also ensured that a name longer than 20 characters would throw an exception with the following code:

*@Test*

void testNullName() {

//Testing null input for name.

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

new Task(“1234567890”, null, “Pick up eggs and milk”);

});

}

To build the JUnit tests for contact service and task service I systematically took one requirement at a time from the contact service and task service bullet points and ensured that specific tests were written for each statement. These classes import their respective contact and task classes, which passed tests of the same nature, and 100% of the listed requirements were tested and passed for these classes. This can bee seen in the following examples of coverage provided below:

A screenshot of a computer

Description automatically generated with medium confidenceA picture containing table

Description automatically generatedA screenshot of a computer

Description automatically generated with medium confidence

The 94.2% coverage was the minimum percentage hit, as the rest were at 100%

To ensure this project holds up to industry standards for efficiency, I structured my code in a concise way that is well documented and easy to read. The constructors have only the essential parameters which eliminates unnecessary code, values, and minimizes the possibility of unintentional behavior. They also have statements to validate input before they are passed to the object, for example the task constructor is made up of the following block of code:

// task constructor

public Task(String idNum, String name, String description) {

// id validation per requirements

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

throw new IllegalArgumentException("Id is invalid!");

}

// ensures id is unique

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

if(TaskService.*taskList*.get(i).getId().equals(idNum)) {

throw new IllegalArgumentException("Id is already taken!");

}

}

id = idNum;

this.setName(name);

this.setDescription(description);

}

Keeping the parameters to a minimum and using structured blocks of statements as opposed to optional parameters and multi-level nested loops and if-statements also adds to the efficiency of the code. Deeply nested code blocks can have undesirable results, can make it hard to trace and correct these issues, and overall make for poor readability in terms of truly understanding the intent of the code.

General test case structure consists of four stages: setup, exercise, verify, and tear down. Stages two and three are where actual testing happens, and it’s in stage three that expected outcomes are compared to actual outcomes with assertions. Assertions are Boolean statements that are used to validate software correctness. (Garcia, 2017)

In the code below I used assertTrue methods to deconstruct the testContact object and verify that the known values passed in the constructor were indeed what the object would return with the mutators.

// valid contact creation

*@Test*

void testContact() {

Contact testContact = new Contact("1234567890", "Jane", "Doe", "9523814409", "123 Elm Street");

*assertTrue*(testContact.getId().equals("1234567890"));

*assertTrue*(testContact.getFirstName().equals("Jane"));

*assertTrue*(testContact.getLastName().equals("Doe"));

*assertTrue*(testContact.getPhone().equals("9523814409"));

*assertTrue*(testContact.getAddress().equals("123 Elm Street"));

}

If any value other than the that which was specified was returned the test would fail. For verifying that values that were passed to constructors that were out of bounds or null, I used the assertThrows method, which will fail if the passed arguments are valid and no exception is thrown. The following code is an example of my work.

*@Test*

void testNullID() {

//Testing null input for ID.

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

new Task(null, "Grocery", "Pick up eggs and milk");

});

}

*@Test*

void testLongID() {

//Testing null input for ID.

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

new Task("12345678901", "Grocery", "Pick up eggs and milk");

});

}

Junit provides repeated testing ability by adding the line of code “#RepeatedTest(n)” to the front of testing methods, where n is the amount of repetitions a given test will be run. (Garcia, 2017). I did not see the need to implement this in any of my test cases but will consider it in the future should the opportunity present itself. Adding this line to a test method will add values for Tests run, Failures, Errors, and Skipped to eh results of the test case.

The assertion method assertTrue is useful in any application where values are being passed and stored to verify that the data will be set and returned as expected. This will verify not only the constructor for the object, but also the access mutators as well.

The method assertThrows can be used in any situation where invalid arguments should throw exceptions, such as passwords not containing the proper amount and style of characters. Meeting these types of security-based requirements is especially critical when designing a solution for a client.

I exercised caution in programming in this project by validating values before they were passed as arguments to constructors, and thoroughly testing the exceptions and error were thrown where appropriate. This can be seen in the code below for the constructor:

// appointment constructor uses Integer class in order to test for null input.

// primitive int defaults to 0 which could be interpreted as January, or first index

public Appointment(String idNum, Integer month, Integer day, Integer year, String description) {

// id validation per requirements

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

throw new IllegalArgumentException("Id is invalid");

}

And below for the test:

@Test

void testNullId() {

// Testing null input for id

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

new Appointment(null, 8, 5, 2022, "This should throw an exception");

});

}

@Test

void testLongId() {

// testing too long Id

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

new Appointment("12345678901", 8, 5, 2022, "This id is 11 char, should throw an exception");

});

}

The discipline in coding came in breaking tasks down into manageable blocks with a full understanding of the functionality as opposed to over-complicating processes. This came into play when validating the appointment date met the defined requirements as shown below:

// date validation

Date currentDate = new Date();

Calendar c = Calendar.getInstance();

// broken down into month, day, year for the purpose of passing only these arguments to the constructor

c.set(Calendar.MONTH, month);

c.set(Calendar.DATE, day);

c.set(Calendar.YEAR, year);

// load them into a Date object

Date newDate = c.getTime();

if(!currentDate.before(newDate)) {

throw new IllegalArgumentException("Date is in the past!");

}

References

García, B. (2017). *Mastering Software Testing with JUnit 5*. Packt Publishing.

<https://eds-b-ebscohost-com.ezproxy.snhu.edu/eds/ebookviewer/ebook/bmxlYmtfXzE2MjY5NTBfX0FO0?sid=c9161a99-3966-45bd-8024-8a17fd0bd325@sessionmgr102&vid=1&format=EB&lpid=lp_94&rid=0>