**CS-320 Project Two**

**Christopher Clark**

**June 15, 2022**

**Summary**

Throughout this project we had three different services with their own system requirements. Once implemented, JUnit testing was employed to confirm the operation of each class and service. We were also able to uncover any bugs that needed correcting. In the contact class the system requirements called for a unique contact ID that was not null and no longer than 10 characters. We also needed first name, last name, and telephone strings with similar characteristics. Finally, there was a requirement for an address string that is not null and no longer than 30 characters. There were similar requirements for the task and appointment classes and services as well. My JUnit testing was designed to capture all possibilities we might encounter. One example is seen below: (Font size set to 8 for code examples throughout this document for space saving).

*@Test*

*@DisplayName("Accepts ID with length of 10") //Pass ID with 10 char*

*void IdWithTenCharTest() throws IllegalArgumentException {*

*Contact contact = new Contact("1234567890", "Jane", "Doe", "1234567890", "123 First St");*

*assertEquals*("1234567890", contact.getID());

}

This test accepted an input for the unique ID that was ten characters long. We also tested for an ID length of one, more than ten characters, and null. We used similar testing for the first and last name strings as well as the telephone string. The same method was used with the address string except the length was tested for one character, thirty characters, more than thirty characters, and null.

Within each test we included variables that would allow the test to pass. In other words, if the test was to throw and exception if the address was more than thirty characters, then we included a test variable with more than thirty characters. This proves the task class was functioning as designed.

When addressing task service class, we used similar testing. However, we changed our approach slightly. We added a test to validate the class by adding an entry to the hash map that included a unique task ID, name, and description as seen below. This allowed me to remove the test for one character as we had done in the contact class.

*@Test*

*@DisplayName("Validate task class")*

*void testAddTask() { //Validate data added to hash map*

*Task task = new Task("1", "Name", "Task Description");*

*assertTrue(task.getTaskId().equals("1"));*

*assertTrue(task.getName().equals("Name"));*

*assertTrue(task.getDescription().equals("Task Description"));*

*}*

From this point we continued to test the task class based on the requirements document. The unique task ID was to be no longer than ten characters and not null. We then tested for the name string which was not to be longer than twenty characters and not null. Finally, the description string was tested as required to not be longer than fifty characters and not null.

The contact service class was required to have the capability to add, delete, and update the contact entries. We used JUnit testing to validate the operation of the code. One example is seen below.

*@Test*

*@DisplayName("Contacts can be deleted") //Verify contacts can be deleted*

*void DeleteContactTest() {*

*ContactService contacts = new ContactService();*

*contacts.addContact("Jane", "Doe", "1234567890", "123 First St");*

*contacts.addContact("Caleb", "Thomas", "2345678901", "234 Second St");*

*contacts.addContact("Caitlyn", "Tyler", "3456789012", "345 Third St");*

*contacts.deleteContact("2");*

*assertFalse(contacts.contacts.containsKey("2"));*

In this example we are testing that contacts can be deleted. There was a total of six different tests used to encompass all of the contact service class requirements.

In the task service class our requirements were very similar to the contact service class. The task service class was required to add tasks with unique IDs, delete the task per ID, and update the task per ID. There were only four tests added since there were fewer variables that could be updated. An example is shown below.

*@Test*

*@DisplayName("Validate task service class")*

*void testTaskService() { //Verify task service operation*

*TaskService taskService = new TaskService();*

*Task task = new Task("12345", "TaskName", "Task Description");*

*assertFalse(taskService == null);*

*assertFalse(task == null);*

In this example we tested the task service class to verify a task with a unique ID, name, and description could be added successfully. We were able to see coverage percentages over 80% for these tests.

The appointment class and service used similar testing for the requirements listed. We had to have a unique appointment ID that was not null and no more than 10 characters long. We also had to provide for an appointment date that could not be null or be in the past. The appointment service requirements included the ability to add or delete appointments per ID. The testing used for this service included JUnit tests for the included requirements. Below is an example of the testing used to verify the appointment ID is not too long. The system should throw and error if true.

@Test

@DisplayName("Throws error if appointment ID is too long")

void testAppointmentIdTooLong() {

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

new Appointment("01234567890", new Date(), "Description");

});

}

The appointment service class was tested with JUnit testing to prove the operation and ability to add and delete appointments based on the ID. An example is seen below:

@Test

@DisplayName("Validate appointment service class")

void testAppointmentServiceClass() {

AppointmentService appointmentService = new AppointmentService();

Appointment appointment = new Appointment("0123456789", new Date(), "Description");

assertFalse(appointmentService == null);

assertFalse(appointment == null);

}

These testing examples functioned as intended and provided coverage of more than 80%. In order to improve the efficiency of the code as originally written, some changes were made. Originally the contact, task, and appointment classes had validation code within the constructors. A change was made to move the validation code to the setter and call the setters from the constructors. This improved testing coverage to nearly 100% for the testing used. Another attempt to keep the code as efficient as possible was to not over test. Basically, there were originally tests for variables that checked for null, over 10 characters, and equal to 10 characters. On top of that other tests had been included for testing within the acceptable parameters for input with 1 and 9 characters. This was overkill as the variable had already been tested for input within and outside of the requirements. Those tests were removed as redundant. This simplified the overall code and improved the readability and coverage.

By simplifying the code we can also reduce the required resources to run the system. It makes for a more technically sound system. Code written with best practices in mind can make for a better understanding for developers that may follow the original team. In turn, future system updates will be less costly and more efficient. By moving the validation code to the setters and calling them from the constructors, the code performs better. Keeping to a standard naming convention for variables, classes, and methods throughout the code also follows best practices. Technically sound code is best accomplished iteratively as well. Testing and validating smaller sections of code rather than an entire system makes corrections much easier. Trying to find bugs in code can be very challenging if your testing hundreds of lines of code instead of just a few. We have demonstrated this here based on the success of our JUnit tests and the coverage we observed.

**Reflection**

Throughout this project we have employed the JUnit testing models for assertions. In contact service test we included assertion methods for *assert equals, assert not equals, assert false,* and *assert throws.* In the task and appointment service tests we used much the same assertions methods in each. We used *assert true, assert false,* and *assert throws* methods*.*

According to junit.org: (Assert, n.d.)

* The *assert equals* method asserts that two doubles or floats are equal within a positive delta.
* The *assert not equals* method asserts that two doubles or floats are not equal within a positive delta.
* The *assert false* method asserts that a Boolean condition is false.
* The *assert true* method asserts that a Boolean condition is true.
* The *assert throws* method attempts to run a piece of code in order to verify an exception occurs.

There are several assertion methods not used in any of the milestone assignments. Some of those are *assert same, assert not same, assert null, assert not null.* There are also annotations used in JUnit testing that were not employed here. Some of those include @Nested, @Disabled, @After, @BeforeEach, @AfterEach.

According to junit.org: (Assert, n.d.)

* The *assert same* method asserts that two objects refer to the same object.
* The *assert not same* method asserts that two objects do not refer to the same abject.
* The *assert null* method asserts that an object is null.
* The *assert not null* method asserts that an object is not null.

In JUnit testing annotations are used to label test methods and configure them. The ones I have identified are defined according to the JUnit 5 users guide below: (Bechtold, n.d.)

* @BeforeEach means the labeled code is executed before each test.
* @AfterEach means the labeled code is executed after each test.
* @Nested means that the annotated class is non-static nested test class.
* @Disabled means test class or method is disabled and should be ignored.

There are several other techniques listed in the users guide. To define them here would add several pages to this report. It is helpful to know that the guides and JUnit resources are available. The assertion methods are very useful for testing user inputs and expected outputs for section of code being tested. Being able to test separate outputs using Boolean methods is useful. Knowing if objects are equal or null though testing is important to the overall outcome of a project. Labeling tests with annotations like @BeforeEach can provide greater detail in testing individual code snippets. Having the ability to annotate what tests are disabled can also provide for better solutions to issues found when performing system testing. The JUnit testing used in these milestones has proven to be a very useful tool and should be a part of every project team’s arsenal.

**Mindset**

Our approach to this project was one of best practices. We took care to make sure the overall mechanics and style of the code remained consistent throughout. We were cautious in our approach to testing the code by using common and well known testing techniques. In this case we utilized JUnit testing. We were careful to ensure that all the requirements were tested thoroughly. This can be seen in the contact test class that was submitted. In the example below we have tested inside and outside of the parameters for the unique contact ID requirement.

@Test

@DisplayName("Accepts ID with length of 1") //Pass ID with 1 char

void IdWithOneCharTest() throws IllegalArgumentException {

Contact contact = new Contact("3", "Jane", "Doe", "1234567890", "123 First St");

assertEquals("3", contact.getID());

}

@Test

@DisplayName("Accepts ID with length of 10") //Pass ID with 10 char

void IdWithTenCharTest() throws IllegalArgumentException {

Contact contact = new Contact("1234567890", "Jane", "Doe", "1234567890", "123 First St");

assertEquals("1234567890", contact.getID());

}

@Test

@DisplayName("ID more than 10 characters throws exception") //Throws exception for ID with more than 10 char

void IdWithMoreThanTenCharsTest() {

assertThrows(IllegalArgumentException.class, () -> {

new Contact("12345678901", "Jane", "Doe", "1234567890", "123 First St");

});

}

@Test

@DisplayName("Null ID throws exception") //Throws exception for null ID

void NullIDTest() {

assertThrows(IllegalArgumentException.class, () -> {

new Contact("", "Jane", "Doe", "1234567890", "123 First St");

});

}

We were also careful not to over test and complicate the code. This will help with future readability for other teams looking to amend or upgrade the system. In using this approach, we are better able to understand how the code is built and in what ways the various services and classes are related. It is important to understand this in order to reduce bugs and errors that may cause failures. When one class calls a public method from another class, these interrelationships must be sorted out in order for the system to function. If a class name is not called out correctly, certain functions will fail. Testing for these calls and input variations is one way to help reduce issues for the client. In the example below, the contact service class is calling a setter from the contact class. If the syntax is wrong, the service will male function.

public void updateFirstName(String ID, String newName) { //Update first name method with error checking

if (contacts.containsKey(ID)) {

try {

contacts.get(ID).setFirstName(newName);

} catch (IllegalArgumentException e) {

e.printStackTrace();

}

}

}

This example has code that was updated to remove the error catch and throw an exception instead. However, the call to the setFirstName method in the contact class is demonstrated.

One issue we took into consideration when testing the code was how to avoid personal bias with regard to our own code. One approach for this is to always remain objective and open to changes. Another method is to allow others to provide input on the techniques employed. Sometimes it is inevitable that a developer will be required to test their own code. In this case we must apply a best practices method and test iteratively and objectively. A developer must be willing to learn from previous mistakes and improve their own processes as they grow. In this case we employed testing that would cover the requirements as they are. Even though the testing was successful, it was discovered that there is a better approach to how the constructors and setters were used. The recommendation to move validation from the constructor to the setter and then call the setter from the constructor was found to improve testing coverage. Therefore, the changes were employed despite the satisfactory result from the previous iteration. Below is the before and after example of this.

**Before**

if (this.validateID(ID)) { //Error check for contact ID valid or print message

this.contactID = ID;

} else {

throw new IllegalArgumentException("ERROR: Invalid ID");

}

**After**

if (ID != null && ID.length() <= 10) { //Error check for contact ID valid or print message

this.contactID = ID;

} else {

throw new IllegalArgumentException("ERROR: Invalid ID");

}

setFirstName(first);

setLastName(last);

setPhone(phoneNum);

setAddress(anAddress);

//\*\*\* Setter

public void setFirstName(String first) throws IllegalArgumentException { //Setting the first name to variable or throw exception

if (first != null && first.length() <= 10) {

this.firstName = first;

} else {

throw new IllegalArgumentException("Invalid first name");

}

}

Here we have removed validation from the first and last names, phone, and address constructors. We are only calling the setters where the validation has been added.

Everything accomplished here required a certain level of discipline. It is important to maintain best practices when approaching a project like this. All ethical and professional ideals should be maintained by all involved. We have all heard of issues that implemented systems have had. For example, in 1999 the Mars Climate Orbiter was lost when approaching Mars orbit. It was found that a simple mistake in the units of measure used by the developer caused the crash. The developer used British units of measure while JPL utilized the metric system when imputing navigational changes. (WP Company, n.d.) Had better, more disciplined testing been used, this could have been avoided. One could argue that this should have been corrected in initial documentation, but four our purposes, we are comparing the issues to how a more disciplined approach is necessary. In the case cited above, it is clear that no stone should be left unturned. By that we mean, all steps should be taken to avoid these types of issues prior to implementation. This must occur within the constraints of the project charter, but professional developers should never allow cost cutting measures that jeopardize the quality and safety of the projects they are involved in. Professional developers are also under an obligation to continually improve their knowledge of their craft. Continued education and process improvement should be a high priority for any professional. If you find yourself technically deficient within your organization, you should take steps to resolve it. Obtaining certifications and professional degrees is one way to ensure technical proficiency now and in the future. This team member is currently pursuing a professional degree in computer science with certifications from CompTIA in A+, Net+, and Security+. We all owe it to ourselves, our organizations, and our clients to be the best we can within the scope of our chosen professions.

**References**

Assert (junit API). (n.d.). Retrieved June 2, 2022, from <https://junit.org/junit4/javadoc/4.8/org/junit/Assert.htm>l

*Assert.throws*. NUnit Docs. (n.d.). Retrieved June 2, 2022, from <https://docs.nunit.org/articles/nunit/writing-tests/assertions/classic-assertions/Assert.Throws.html>

Stefan Bechtold, S. B. (n.d.). JUnit 5 user guide. Retrieved June 2, 2022, from <https://junit.org/junit5/docs/current/user-guide/>

WP Company. (n.d.). The Washington Post. Retrieved June 13, 2022, from <https://www.washingtonpost.com/wp-srv/national/longterm/space/stories/orbiter100199.htm>