**CS-320 Project Two: Summary and Reflections Report**

**By David Shelt**

**Southern New Hampshire University**

1. **Summary**
2. **JUnut Testing Approach**
3. **Alignment to Software Requirements**

The testing approach that I used was to first set the requirements I was given in the prompt and guidelines by making Illegal Argument Exceptions inside of the constructors that I created in the Contact.java, Task.java, and the Appointment.java files. This eventually became the ProjectOne.java file when I merged the files together from the assignments. I ensured that Illegal Argument Exceptions were thrown if the number of characters in the variables exceeded the limits that were given, or if the variables were null. So, for example, in the ContactTest.java, TaskTest.java, and AppointTest.java files, I created a test that would throw an exception if the contactId, taskId, and apptId variables each exceeded 10 characters in length. Then, when I went into test the contactId, taskId, and apptId variables, I entered more than 10 characters in for each of the variables in their respective tests. This made the exception occur so that it would meet the software requirements by making sure none of the characters exceeded 10 characters in length. Also, I set up tests to ensure that new Ids would not be entered for the service test and made sure that tests passed for the false parameters that I set up, such as adding the same Id characters for the contactId, taskId, and apptId variables to make sure they could not be entered twice into the list.

1. **Effectiveness of JUnit tests**

I am glad I learned about JUnit tests in module three. I had no idea of what Junit testing was initially. I thought it was simply just running the code and putting in different inputs and that was it. I had no idea what was involved in JUnit testing. Luckily, there was a video supplied to us as a tutorial, which helped me understand what I needed to do going into the assignments and project. Each assignment had a different coverage percentage For example, there was more to cover in the contact class than there was in the task class, and a little more to cover in the appointment class than the task class. This was due to the contact and appointment classes requiring more parameters than the task class. Also, all of the service tests had a higher coverage because they needed more functions, such as adding, when possible, along with deleting and updating the contacts, tasks, and appointments, as long as they met the parameters set in the contact, task, and appointment files. I made sure that the test passed and one great things about using JUnit testing is when you have errors or failure, it will tell you where to look to address those errors. Overall, I personally found JUnit testing to be very useful and I intend to use this type of testing more in my academic and professional careers when using Java in particular. If there is something like this in Python, C++, or any other languages, then I intend to use those as well.

1. **Experience with JUnit Testing**
2. **Ensuring Code was Technically Sound**

One thing that I did was use data structures such as arrays in order for the contact and task list to be stored. One was in lines 7 and 11 in the ContactService.java file where the list would store the contact info with the contactId, first name, last name, phone number and address and then created a constructor with a new array list in line 11. It looked like the following:

private ArrayList<Contact> contacts;

//Creating a new constructor with a new array list for contacts

public ContactService() {

contacts = new ArrayList<>();

}

I did the same in line 8 of the TaskService.java file, followed by making the constructor with a new array list in line 12.

private ArrayList<Task> tasks;

//Creating a new constructor with a new array list for the tasks

public TaskService() {

tasks = new ArrayList<>();

}

Also, for the AppointmentService.java, I also added a new array list along with a constructor with the that contained the appointment ID, appointment date, and description. Below is the code shown in the AppointmentService.java file for lines 8 through 13.

//Making the list of appointments

private ArrayList<Appointment> appointments;

//Creating a new constructor with a new array list for the appointments

public AppointmentService() {

appointments = new ArrayList<>();

}

In the TaskServiceTest.java, ContactServiceTest.java, and AppointmentService.java file,s I made assertations for the files to come out either true or false to make sure the tests ran correctly. This was from the TaskService Test.

public void testMethodAddPass() {

TaskService cs = new TaskService();

Task c1 = new Task("T123", "Doing the dishes", "Wash, dry, and put away dishes");

Task c2 = new Task("T213", "Take out garbage", "Remove full garbage bag, place outdoors");

Task c3 = new Task("T312", "Do the laundry", "Wash, dry, and fold clean clothes");

assertEquals(true, cs.add(c1));

assertEquals(true, cs.add(c2));

assertEquals(true, cs.add(c3));

}

This was from the ContactServiceTest:

@Test

public void testMethodAddFail() {

ContactService cs = new ContactService();

Contact c1 = new Contact("C123", "Douglas", "Smith", "6315557898", "1313 Mockingbird Lane");

Contact c2 = new Contact("C213", "David", "Shelt", "6315555355", "1313 Mockingbird Lane");

Contact c3 = new Contact("C312", "Christina", "Rose", "6315559845", "43 Fifth Ave");

assertEquals(true, cs.add(c1));

assertEquals(false, cs.add(c1));

assertEquals(true, cs.add(c3));

assertEquals(true, cs.add(c2));

}

This was from the AppointmentServiceTest:

//Creating a failing method for adding the appointments

@Test

void testMethodAddFail() {

//Putting in the date

Date apptDate = new GregorianCalendar(2022, Calendar.FEBRUARY, 25).getTime();

AppointmentService cs = new AppointmentService();

//Creating the appointments

Appointment c1 = new Appointment("A0000001", apptDate, "Change Oil, Rotate Tires");

Appointment c2 = new Appointment("A0000002", apptDate, "Change Brakes");

Appointment c3 = new Appointment("A0000003", apptDate, "Perform Engine Maintenance");

//Putting in passing appointments and trying to add an existing appointment to make sure it fails

assertEquals(true, cs.add(c1));

assertEquals(false, cs.add(c1));

assertEquals(true, cs.add(c3));

assertEquals(true, cs.add(c2));

}

1. **Ensuring the Code Was Efficient:**

When I first started the work on what became the first part of the project in module three, it took me two days and I wrote way too much code at first to where it was just a mess and became loaded with errors or conflicts after running the JUnit tests. Also, I had a few trial and run errors that I had until I deleted everything, looked up all the resources to help me write the code, and started again from scratch. What I found was sticking to the basics in the Contact.java, Task.java, and Appointment.java files really helped. I created the variables and then made the Illegal Argument Exceptions if the variables exceeded certain lengths, such as 10 characters for the contactId, taskId, and apptId variables for example. Then, I created the this.method to take sure the variables would update for the contacts. I also put in the getters and setters for all the variables for contacts, tasks, and appointments. For example, this is what I did for the Task.java file.

package taskServ;

public class Task{

//Creating the variables for the class for task ID, task name, and task description

private String taskId;

private String taskName;

private String taskDescrip;

//Creating a constructor with parameters for the variables

//Making throw IllegalArgumenExceptions for each variable

public Task(String taskId,String taskName, String taskDescrip) {

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

throw new IllegalArgumentException("Invalid Task ID");}

if(taskName == null || taskName.length()>20) {

throw new IllegalArgumentException("Invalid Task Name");

}

if(taskDescrip == null || taskDescrip.length()>50) {

throw new IllegalArgumentException("Invalid Task Description");

}

this.taskId = taskId;

this.taskName = taskName;

this.taskDescrip = taskDescrip;

}

//Creating the getters and the setters for the variables//

public String getTaskId() {

return taskId;

}

public String getTaskName(){

return taskName;

}

public void setTaskName(String taskName) {

this.taskName = taskName;

}

public String getTaskDescrip() {

return taskDescrip;

}

public void setTaskDescrip(String taskDescrip) {

this.taskDescrip = taskDescrip;

}

}

I also made sure that the testing put in was to the point and through assertions to make sure we know whether the variables inside of the constructors passed the tests or not. This was from the TaskTest.java file.

class TaskTest {

// Testing with all variables inside set parameters

@Test

public void createValidTaskData() {

Task task = new Task("T000000001", "Doing the dishes", "Wash, dry, and put away dishes");

System.out.println(task);

}

//Testing with invalid taskId

@Test

public void testTaskIdTooLong() {

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

new Task("T000000000000001", "Doing the dishes", "Wash, dry, and put away dishes");

});

}

// Testing with invalid taskName

@Test

public void testTaskNameTooLong() {

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

new Task("T000000001", "Doing the dishes, washing the dishes, storing the dishes", "Wash, dry, and put away dishes");

});

}

//Testing with invalid taskDescrip

@Test

public void testTaskDescripTooLong() {

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

new Task("T000000001", "Doing the dishes", "Wash the dishes with soap, rinse the dishes with water, dry the dishes with a cloth, and put away dishes in cupboard");

}); }

}

Notice how in the test called testTaskIdTooLong, I put in more than 10 characters for the task Id number. That was all that needed to be entered to make sure the exception was thrown. I also did the same thing by adding too many characters in the taskName and taskDescrip variables and testing them to make sure exceptions there thrown. Therefore, I made sure the tests were to the point and not too elaborate or sophisticated at all. Also, this helps those who will read and change code in the future when needed will know what I did and be able to make simple adjustments to the code rather than taking a long time to change it or getting lost.

1. **Reflection**
2. **Testing Techniques**
3. **Testing Techniques Used**

From modules three to five, I conducted testing on a contact class, contact service class, task class, task service class, appointment class, and appointment service class. For the contact, task, and appointment classes, I had to make sure that the variables I created fit the parameters that were set in the requirements. What this meant was I needed to have the amount of the characters in each of the variables at a certain length and for the appointment class, I had to have the date set no earlier than the current date for the appointment class, which was a little difficult for me in the beginning.

The technique I used was more of testing the functionality by using the assertion methods. An assertion is a Boolean statement used to reason about the correctness of software being developed (Garcia, 2017). For all three classes of Contact, Task, and Appointment, I created if statements to throw illegal argument exceptions if the values of the variables I created were null or exceeded a certain length of characters. This verified if an exception is raised in a piece of code (Garcia, 2017). If it was null or was too long in length, then it would tell the user that the value for the variable they were trying to enter was invalid. Then I put in getters and setters for the variables.

For the ContactService, TaskService, and AppointmentService classes, I created new array lists and then I made boolean methods with for loops with if statements to check if the contacts, tasks, or appointments could be added to the list. If the inputs were already there, then the users would be notified. If they were not in the respective arrays or lists, then they could be added to what the user wanted. If the items were there already, then they could be removed or updated if needed.

For the ContactTest, I made a constructor with the respective variables and then used the assertTrue method with the variables that fit the parameters that were expected. This method determines if the Boolean expressions is true (Garcia, 2017). I then made a print statement to print them out. Then I made another test with the characters in the ID numbers being too long to see if it would fail by putting in an assertion throw. This was followed by creating a test with the characters for the first and last names being too long. Also, there were tests with the characters in both the contact number and contact address being too long. Upon testing, all the tests passed. For the service tests, I would use the assertEquals methods to test whether I could add, delete, or update the contact, tasks, and appointments. I also tried out in false statements to make sure there would be failures. Upton testing, all the tests passed.

1. **Testing Techniques Not Used**

There could have been other methods I could have used, but I decided not use those techniques because I felt it could have made the testing more complex, where this course has taught me to keep things simple. Upon briefly looking up the different techniques that I did look into, I determined that it would be too much and did not fit what I was looking to do overall since this was a rather simple project in terms of testing. One method I saw was using assertTimeout, which make an exception happen if a certain amount of time is exceeded before an execution is completed. We can set the time for any time we wish to, but the normal time is 2 minutes (Garcia, 2017). Another technique is an assertDoesNotThow, which makes sure that an exception is not thrown for the given executable code. Another method I saw is dynamic testing, also known as test factory testing and cannot be private or static.

1. **Practical Uses and Implications**

I started with a throwing an illegal argument exception, which is to be used when making sure the parameter or requirements we need to have pass through testing, and to let the user know that if they do not fit within the parameters or requirements that were set. assertDoesNotThrow does the opposite of this and should be used when we need executions to go through always. assertEquals should be used when we need to determine whether two objects are equally defined or not. assertTimeout should be used if we are either requesting or receiving data, that way the execution will stop when the time is exceeded, which is usually 2 minutes (Garcia, 2017). Dynamic testing or test factory should be used when we are working with arrays of files or streams, which in this case was not necessary but could have been had the requirements been more complex.

1. **Mindset**
2. **Employing Caution**

I believe that I employed a great deal of caution while working on the milestones and the assignments. When I first approached this assignment, I originally tried other methods to try to make my code fancy. All I ran into was errors. Then I thought to myself, “What the heck am I doing here?” I went back and read and reminded myself that I need to keep it simple, not just for others who were reviewing the code, but also for myself. I also had to keep in mind that I was working on multiple files and therefore had to be simple with the code to make sure that the files could work off each other, such as the Contact.java file and the ContactTest.java file.

It is necessary to appreciate the complexity and interrelationships with the code. In fact, it should be embraced by us as developers and testers. We must keep in mind that any small mistake can ruin the entire project and it will be tough to find the mistakes that were made, which could be cost and time consuming. For example, when I was working on and testing the taskId, I had to make sure that the characters were not null and did not go over 10 characters in length. That is why I put in a throw exception in there.

1. **Limiting Bias in Code Review**

I knew from writing the code there could be a little bit of bias from me in testing to wanting to code to pass no problem. Nobody wants to see their code they developed fail, and it’s a hard pill to swallow sometimes. But it is a harsh reality we must address. The simple way I went about doing that is letting the JUnit tests perform the duties. From there, when I got errors in my tests, the results would tell me what I needed to fix or address. In a way, the JUnit tests made itself the independent testers and told me where I could improve. For example, in the Appointment.java and AppointmentTest.java, I was having problems with the dates, and it kept failing with may approaches that I took, even though the code was clean. I was able to then use the Gregorian calendar function and that was able to make the tests pass when I wanted then to pass, and fail when I put a past date in. Had I not performed a JUnit test, this would have been an error that was undetected and could have caused problems after deployment.

1. **Being Disciplined to Quality**

It is extremely important to be disciplined in our software testing. We are the Quality Control people in the software development world. If we let a bad product go through and be deployed, then the responsibility will fall on us as testers, and we will have to own it. If the damage is catastrophic, then this could hurt our reputations in our careers. This means nobody will want to hire us if we decide to cut corners and be sloppy with our testing. Even worse, this could result in the loss of lives as many software problems have led to airplane crashes in the past, such as the Boeing 737 Max crashes a few years back.

This means as a quality tester, I must leave no stone unturned and make sure all the requirements are met and they pass successfully, even when trying to find errors or failures in the code. This means I need to be better at writing code, especially when I need to write complex code rather than simple code. Having this discipline will ensure that I will have a successful career in the world of computer science, no matter which route I ultimately choose to take as my career.

**Reference:**

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