Matthew Stebbins

Project two

CS-320-T2822

**Summary**

Going into testing with JUnit I was worried about how hard it would be to get a high coverage. The Contact service required a new class with five variables with differing requirement. Some could not be changed, none could be null, most had a max length. Along with the class there was also a service class to support it and to be the interface to the contact class. This is what will allow the system to add, delete, and update the contacts from the contact class. With JUnit I was about to start by testing the happy path and then testing each requirement making sure the right errors are thrown when I triggered them. Example being that each variable in the Contact class could not be null so for each variable I tried creating a object that had one of the variables set to null to see if it would trigger the error.

As I mentioned above, I was worried about getting a high coverage percent but with JUnit and Eclipse there is an option the let me see a percentage as well as a color-coded display of the coverage. In the end I had a 100% coverage on the Contact class and a 96% on the service. With a total coverage across the three services being 83.7%. This leads to a high confidence that the code will function as intended.

A good example of sound and efficient code is in the contact class I used the seter functions for the constructor to set the starting values.

public ContactClass(String ID, String firstName, String lastName, String phone, String address) {

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

throw new IllegalArgumentException("Invalid ID");

}

this.ID = ID;

this.setFirstName(firstName);

this.setLastName(lastName);

this.setPhone(phone);

this.setAddress(address);

}

This reduced the amount of redundant code and the number of tests needed to check all error paths.

**Reflection**

* **Testing Techniques**

For the last three weeks we have been learning and implementing JUnit tests for three different services. The services were Contact, Task, and Appointment. Each had their own sent of requirements to create and test. The first week was creating the Contact service and having never used JUnit everything had to be built from the ground up one unit at a time. The Junit Tutorial (Junit tutorial 2020) was a great help at this point showing me a working example of what it should look like. For these services I used both unit and integration tests to check that they fulfill the requirements. An example of a unit test is testAppointmentClassIDToLong() here JUnit is testing to see if the AppointmentClass() was implemented to the requirement that IDs cannot be over ten digits long. This isolates that one requirement and tests it as its own unit.

An example of an integrated test would be testAppointmentServiceClassIDAlreadyUsed(). Here the test is testing both the Appointment class as well as the AppointmentService class making sure they function together to provide the requirements as requested.

I was not able to do any Functional, Regression, or Acceptance tests. All three services still need their frontends created since functional test and acceptance tests depend on the end users’ experience. Regression tests are for integration of new functionality to a currently released build and currently we do not have a previous release. (Team, Testing methodologies – five core components of testing)

* **Mindset**

Well working on these services, I had to take on a different mind set then what I have used previously when developing code. Here I wasn’t just developing code and changing things till it worked. I had to think about making my code with the fewest branches and flows leading to simpler code that was easier to test. This led to less redundant code and a faster services in the end. Sometimes you have to have branches and one of these times for me is in the AddAppointmet class.

public void AddAppointment(String ID, Date AppointmentDate, String Description) {

if(Appointments.isEmpty()) {

Appointments.add(new AppointmentClass(ID, AppointmentDate, Description));

}

else {

int i = 0;

while(i < Appointments.size()) {

AppointmentClass temp = Appointments.get(i);

if(temp.getID() == ID) {

throw new IllegalArgumentException("ID already used");

}

i++;

}

Appointments.add(new AppointmentClass(ID, AppointmentDate, Description));

}

}

Here by checking if there are no appointments, we are handling that special case that we now need to test for.

For testing I tried to focus on data to remove as much bias as possible. Using the percentage covered and the visual colors I could see what file needed more coverage and where in that file was not green. By sticking to the hard data and not letting myself decide if any line of code was more or less important then another, I remained disciplined in making sure the services and the tests to check them covered the requirement as they were intended. If I hadn’t it could have led to appointments happening before, they even were added. Contacts that are deleted since the ID was overwrote.

**References**

YouTube. (2020, September 11). *Junit tutorial*. YouTube. Retrieved November 26, 2022, from https://www.youtube.com/watch?v=KbXhK9HUng4

Team, T. nF. (n.d.). *Testing methodologies – five core components of testing*. nFocus Blog. Retrieved November 26, 2022, from https://blog.nfocus.co.uk/testing-methodologies-five-core-components-of-testing