CS 320 Reflection

Thomas Ziesemer

21OCT23

1. **Summary**

During the Contact feature I had just started this kind of testing. While I aligned pretty well with the requirements by making sure that you could not enter two contacts with the same ID, any null inputs, or inputs over character limits I neglected to test for inputs that were exactly at the character limit. For example I made sure that any input for the firstName variable couldn’t be more than 10 characters or null. When testing though I tested a input that was less than the character limit, one that was over the character limit and a third that was null but none at the character limit. At that point my coverage was also a little lacking at only 85%.

The Task feature was very similar to the contact feature but much easier as it had less variables. This one I met all the requirements after learning of my missed parts in the previous feature. The first time I ran the coverage it was at 95%. I found that I had not tested all the if statements in one of the functions so I adjusted the related test adding an extra assertThrow to check the last if statement.

The appointment feature added in the Date object. This was actually more difficult to incorporate than I thought. At first I thought I had it but part the way through setting the date variables I found out that the old date constructor did not work properly anymore outside the empty Date newDate = new Date(); So I had to do some research to find that you could use the Calendar object to create a date and then assign it to a Date object. Once I figured that out I ran into one more issue. Turns out that while you can create a Date and Calendar object outside of a function you needed to create Calendar instances inside the functions to make them work properly. After all of that I was able to finish the rest of the coding smoothly. Testing worked fine and I even got 100% coverage on my first try.

Ensuring the code was technically sound by testing that it did not break down when encountering invalid inputs such as null inputs like in this testing for the appointmentService class:

//Here we will test that going over Char, past dates, non unique IDs, and null inputs throw exceptions

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

Services.addAppointment(underCharID, goodDate, underCharDescription);

}, "Using an already existing ID didn't throw exception");

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

Services.addAppointment(overCharID, goodDate, underCharDescription);

}, "Using over char ID did not throw exception");

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

Services.addAppointment(nullInput, goodDate, underCharDescription);

}, "Null ID did not throw exception");

In addition, the eclipse IDE gave warnings and errors if it found code to be broken or unadvisable. Using those warnings I did research to find better ways to accomplish my coding goal.

When it came to ensuring my code was efficient that was harder to prove. I tried finding if I could do the same things with less lines of code by creating a search function given that three other functions needed to search through the same list but then use the result for different things:

**public** **int** searchAppointments(String appointmentID) {

//If found returns the index of the appointment TZ 29SEP23

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

**if**(Appointments.get(i).getAppointmentID() == appointmentID) {

**return** i;

}

}

//If not found then returns -1 TZ 29SEP23

**return** -1;

}

Here I return the index of the found list item or a -1 to indicate it wasn’t found. For the add function it lets the system know that the ID already exists so a duplicate isn’t made, for the delete function it either uses the index to remove the correct list item or lets the system know that the list item doesn’t exist to delete. Other ways I tried to be efficient included creating variables at the start of the testing instead of typing out each testing input individually which also helped with the technical soundness by making sure that no error happened where I typed the input or the expected output incorrectly when typing it out multiple times.

1. **Reflection**

I mainly used four testing techniques for the project. First I used the equivalence partitioning technique to group testing inputs into four main groups: Null inputs, under character limit, equal to character limit, and above character limit. All inputs of each type should have the same response with some assumptions. Next I used statement testing which is the process of trying to run tests that execute as many statements as possible to cover all the code. This goes well with the decision testing where you check all outcomes for a decision statement such as running all if statements. To test that I was successful in the statement and decision techniques I check the coverage of the code.

There were also several techniques I didn’t use such as decision table and state transition testing which tests how multiple decisions effect future decisions and how changing states changes outputs respectively. I also didn’t do any real boundary value analysis even though it could have possibly been useful. In this type of testing you check the inputs right before, on and after an upper or lower boundary to make sure that there are no weird exceptions close to the boundaries. I did check the boundaries themselves though. Finally, I didn’t have any previous experience with this to lean on to utilize the experience testing technique.

While creating the specific milestones, and later adjusting them for the final project, I tried to keep an open mind on what issues may arise. In the past I found that it is easy to get tunnel vision on what could be the cause of any specific issue and miss other obvious answers. I tried to keep in mind all the normal ways programs mess up and methods to see where a program failed. For example if I was getting failures in my testing then I would create print statements at different parts of the code to figure out where the last place the run reached before it failed out or see if the run entered the correct if statements. Using this with print statements that would make sure the right data entered a function helped narrow where and what the issue was.

While coding it is easy to think you have all of your bases covered using your preferred methods but that is hardly ever the case. To limit how many flaws I may have missed due to my own bias I made sure to run coverages. Also I took on board the critic from my professor when an issue I missed was pointed out. For example in my first milestone I failed to check if inputs the were equal to the character limit would still work. Originally I had checked that inputs below the character limit would work and that inputs above or null would not work. I thought that covered it well but the professor made a good point that if someone would have done an if statement that allowed only less than the character limit then all inputs equal to the character limit would fail even though they shouldn’t. Bias can often cloud our judgment as developers and it can lead to flaws that would have been easily avoided with some sort of outside check.

Limiting bias is one of several things one should do as a developer in their commitment to well written and functional code. Integrity is something taught emphasized in the military and part of the definition is the point of holding true to values even when others cannot see you. As a good developer you should not cut corners in your code and especially not in your testing. Taking the easy way such as just testing one good input and one bad input may get it through QA but the first time it sees production and dozens to millions of users start interacting with it all of the flaws you missed will be found. Sometimes this may only equate to a frustrated user but in the worst cases this could end up with millions of dollars lost or even worse lives.