Simeon Patton

Homework 5 – Test Design Techniques

CS362 Spring 2021

Question 1;

i) Valid Classes and explanation
Valid partition for a date program is going to be *int* [1 → 31] because these are the only values that are going to be accepted when looking at a standard calendar.

ii) Invalid classes and explanation
Invalid partitions for a date program are going to be (-∞ → 1) and (31 → ∞). First, we are going to be looking at -∞ up to and not including 1, these values are not included in a standard calendar because there cannot be the inclusion of a negative day. Next, values ranging from (but not including) 31 to ∞ are also invalid because there is not a month in the standard calendar that is larger than 31 days.

iii) Boundary Values below and above the boundaries
BVA is a lot like partition testing except we are only looking at the values that are surrounding the boundaries of the partitions. This is because errors are more likely to present themselves at or near the boundaries more than within the middle of the partitions.

Because we are looking at a date range program;

Valid Test Cases:	Invalid Test Cases:
Enter the value 1	Enter the value 0
Enter the value 2	Enter the value 32
Enter the value 30	
Enter the value 31	

Question 2:

i) Valid Classes and explanation
Valid partition for a username program is going to be *char* [7 → 10]. This is because a username is going to be characters and can be from 7 to 10 (inclusive) characters long, according to the brief.

ii) Invalid classes and explanation
Invalid partitions for a username program are going to be (-∞ → 7) and (10 → ∞). First, we are going to be looking at -∞ up to and not including 7, these values are not included because the brief was specifically asking for usernames that are at least 7 characters long. Also, there would be no such thing as a negative count username character. Next,

values ranging from (but not including) 10 to ∞ are also invalid because once again, the brief stated that the program was looking for usernames up to a maximum of 10 characters.

iii) Boundary Values below and above the boundaries
BVA is a lot like partition testing except we are only looking at the values that are surrounding the boundaries of the partitions. This is because errors are more likely to present themselves at or near the boundaries more than within the middle of the partitions.

Because we are looking at a program that accepts username lengths of 7→10;

Valid Test Cases:	Invalid Test Cases:
Enter a username of 7 characters	Enter a username of 6 characters
Enter a username of 8 characters	Enter a username of 11 characters
Enter a username of 9 characters	
Enter a username of 10 characters	

Question 3;

i) Valid Classes and explanation
Valid partition for a program that takes in ages is going to be *int* [16 → 60) ∩ (65 → 70]. The reason that we are looking at an intersection (∩) here is because the brief stated that the program will take in [age] values from 16 to 70, except 60 to 65. Because of the inclusion of the keyword "except" we are now looking at a multiple valid partition scenario.

ii) Invalid classes and explanation
Invalid partitions for an age program are going to be (-∞ → 16), [60 → 65], and (70 → ∞). Once again, we have a tertiary invalid partition because the program brief stated that we needed to exclude any age value that was 60 through 65.

iii) Boundary Values below and above the boundaries

BVA is a lot like partition testing except we are only looking at the values that are surrounding the boundaries of the partitions. This is because errors are more likely to present themselves at or near the boundaries more than within the middle of the partitions.

Because we are looking at a program that accepts ages from 16 to 70, excluding 60 to 65;

Valid Test Cases:	Invalid Test Cases:
Enter an age value of 16	Enter an age value of 15
Enter an age value of 17	Enter an age value of 60
Enter an age value of 58	Enter an age value of 61
Enter an age value of 59	Enter an age value of 64
Enter an age value of 66	Enter an age value of 65
Enter an age value of 67	Enter an age value of 71
Enter an age value of 69	
Enter an age value of 70	

Question 4;

i) I believe that unit-testing really favors white-box testing. This is because someone would likely have access to the program code if they are attempting to write tests for each individual “unit” that is found within the application or program itself. It would be extremely difficult to write a test for a portion of the program that you do not know what it is supposed to do. At this point, I have to clarify that it is NOT impossible to write a unit test for a portion of the code that you do not know. This could be accomplished by testing every conceivable input into the unit. This would basically be a brute force black-box style testing that would significantly strain the testing system and would be a very inefficient way to go about it.
All in all, Unit-testing is much better suited to white-box testing, or where the person has an idea of what the code is trying to accomplish so that they are able to write a test to that specific chunk of the code.

Black-box testing (in the context of the Facebook social media platform) would be something like testing the messaging system with repeated messages sent to different recipients to make sure that the correct recipients were receiving the correct messages. This would be considered black-box because I would not have access to the code nor know exactly what the code (within the messaging system) was telling the program to do. We would just be testing to make sure that the “idea” of the messaging system was functioning correctly.

On the other hand, an example of unit-testing with the Facebook platform would be something like testing the input length of the messaging system itself. There would be really no way to accomplish this without seeing the code and determining what (if there even is) a hard coded length limit to the messages themselves.

Question 5;

i) Within the Facebook code, I believe that the places where unit-testing would really shine would be the search function, friend adding, and the marketplace. First, I chose to focus on the search function within Facebook. This is because I believe that it would be pretty strait forward to write tests for the search function. You would be giving terms to search by and then observing the output. This could be accomplished by random inputs in the PyUnit framework as the developer isn't necessarily worried about the actual output of the testing.
Next, I would look at implementing a unit-test within the friend system. This test would look at the correctness of the friends that were added. Basically, the test would be testing the button press on each profile and making sure that the correct end result was given.
Last, I would see about testing the marketplace itself. This is were personally I have most of my issues as this is where I spend most of my time. I would be testing the location values for the searching within the marketplace. This always seems to be incorrect when I am looking at items. I would use a unit-test here coupled with some type of GPS/IP location spoofer to test multiple locations and that the correct location showed up when it was selected.
All of these tests would benefit from an integration type testing framework. This is because it would be very difficult to test the entire system together because there is so much going on with the platform. There would be a much higher bug catch rate when each system is tested individually and then would be integrated back into the system as a whole.

ii) Black-box testing within Facebook could be focused on items that lend themselves to boundary value analysis. On Facebook this would be something like input validation when a user is attempting to create an account and they are looking to put a phone number into the system. This should only work with a value of 10 integers (in the US). Along with the BVA, a generalized input validation would be a good test to run. This is a great black box testing technique because we would have no need to have the source code. We would just be testing the input boxes throughout the platform to make sure that they are taking the correct data type and transferring the information that was inputted to the correct place.

iii) Version control would help out like it always does and provided a basis to roll back to if an update kills the system or an individual unit of the system. This would allow individual developers to work as a team and update each part of the system in smaller groups before integrating it into the Facebook system as a whole. I still believe that the spiral methodology would be best here as the system is already bult and at this point we would be looking at just adding some spirals to the system. Each individual spiral added would include its own development phase and project.