Practical W05-JUnit Student ID: 200007413

JUnit Testing Design

My very first decision was to split the testing procedure into relevant modules which lined up with the interface classes: The factory tests, the loyalty card owner tests, the loyalty card tests, and then the loyalty card operator tests. I began with the factory tests as they doesn't rely upon functionality of the others except insofar as their constructors and setup of initial attribute values. Also, the other tests heavily required the use of the factory to get instances of the relevant classes.

The order of the other classes was chosen based on their reliance on the other classes functionality: owners don't require any functionality from other classes; cards only rely upon the functionality of the owners, and operators require all other classes to be fully functional for most of it's functioning.

JUnit Factory Tests

Here, I wanted to make sure I wasn't testing the functionality of the classes the factory was initialising. Instead, it was just making sure that the factory created not-null instances of the relevant classes when given good input.

I also added in test cases for invalid inputs into this section. For the LoyaltyCardOwner, I decided here that only strings which were valid emails should be accepted, and so when implementing this lead me to not only checking for the invalid but possible null values which had their own set of tests, but also used a standard regular expression for matching emails. I decided to store these as simple string attributes for later retrieval functionality.

JUnit LoyaltyCardOwner Tests

I next went to designing the tests for the LoyaltyCardOwner class. This class doesn't rely upon any of the other classes, as it just holds a name and email. Since I had already implemented the name and email checking and attribute setting, these tests were simple in that I just added checks that the name and email attributes could be retrieved.

JUnit LoyaltyCard Tests

After, I moved onto the cards themselves. Since the factory tests had made sure that an owner instance was passed into the constructor but didn't make sure the owner was set correctly, I made a test here to make sure that the owner instance could be retrieved.

Here, I added tests for retrieving the number of times the card had been used by using the getNumberOfUses() function. This lead me to adding in the integer useCount attribute which could be incremented when points are used. I also checked that when initialised this value is 0.

I also added a test for retrieving the total number of points, which led me in implementation to adding the integer attribute points. I also checked that when initialised, this value was set to 0.

The next bit of functionality needed was to add points. As actually calculating the points earned is something done at the operator level, I just added in a test to make sure that adding positive numbers of points worked. I also wanted to make sure that nothing broke when 0 points were added. Finally, I

added a test for when negative points were added. So, in implementation, I made sure to check the points were greater than zero before trying to change any values.

Next I needed functionality here for using the points, which just meant adding tests for when you have enough points, when you don't have enough points, and for when your points are the same as the points you want to spend. I also then added tests to make sure that once points are used, the useCount incremented. However, I added tests to make sure that when 0 points are used, the counter should not be incremented which led me to adding an additional check in implementation which skipped the incrementation, and points deduction when points used less than or equal to 0 which solved all of the previous edge-case issues.

LoyaltyCardOperator Tests

As this class had the most amount of functionality, and drew together the functionality of all of the other classes, I added these tests last.

I began by checking that owners could be added, including checking that correct exceptions are thrown when an owner is already registered. In writing these tests, I decided that the most logical way to determine if an owner was already registered would be based on email, instead of instance. This was because I didn't want someone to be able to create duplicate owner instances and to register multiple times. This affected the implementation greatly as it meant I needed a way of checking registered owner's emails. To do this in an efficient way, I made the registered_owners attribute for the operator, which would hold all of the registered owners to check for duplicates, and for later retrieval. I made this attribute a HashMap<String, IloyaltyCard>, where the keys are the emails for the owners.

I made the HashMap values the loyalty cards instead of the owners, because the cards had a reference to their owner, but there's no reference to the related LoyaltyCards from the owner. When a user is registered, a unique LoyaltyCard instance is created here and added to the HashMap.

For registering, because I added in tests for registering twice, I thought about how to check for already registered owners in the HashMap. This lead me to using the putIfAbsent() method which would return null if the key was not already used, and would return the current value if it was. Therefore, I could just try to register the owner, which would not overwrite anything, but I didn't have to waste processing time searching the keySet.

I then added in tests for unregistering successfully and unsuccessfully. This meant that later I could add tests for trying to get attributes from unregistered owners knowing that the de-registration was successful and not have to worry about this functionality.

For processing payments, I added similar outlier cases to in the LoyaltyCard tests because it was basically calling those functions and adding some functionality on top. Therefore, along with those tests, I added tests to make sure that points earned from payments were calculated correctly here, as well as checking if a card had enough points when being used.

Finally, with the points functionality added, I could add in tests for the checking mostUsed, and totalPoints. Having implemented reliable functions for retrieving the number of uses and points, I could

just loop over the registered users and determine these values this way which made the process extremely quick.