Overview

For this project I was asked to develop an implementation of a simple loyalty card system described by a set of interfaces. Some choices of the implementation were left up to me, such as the response to certain edge case inputs, but most of the functionality was pre-defined in the interfaces. I needed to use a Test Driven Development process while creating my program, and ensure that all functionality of my program was being effectively tested. I took example from the sample test provided with the project brief, and built my test suite method by method from the interfaces, including both central and edge cases.

Design & Implementation

All of my JUnit tests are held within the 'Tests' class. I began testing without a 'before' test, as I needed to develop the tests and functionality for creating instances of classes (through the factory) before I could rely on this functionality for further testing. I began by creating factory tests similar to the one provided in the brief for each of the other two classes – LoyaltyCardOperator and LoyaltyCard itself. After confirming that these instances had been created successfully using 'assertFalse', I began testing on the methods defined by the ILoyaltyCard interface.

LoyaltyCard

'getOwner' is very simple to test, as the card's owner is defined in the constructor for an instance of a loyalty card. The owner added in the test must simply be the same as the owner returned via 'getOwner'.

'addPoints' is one of the core functions of a loyalty card, yet it is still relatively easy to test with the use of the 'getNumberOfPoints' method. Both of these methods are tested simultaneously, as they must be consistent with one another. The edge case of adding a negative number of points must be refuted by the implementation, as removing points can only be done through spending them. Adding zero points must also have no unintended effects, and finally the card must be confirmed to have zero points upon initialisation of the instance.

'usePoints' is the other core functionality of the loyalty card, but the implementation is slightly more complex, as the 'InsufficientPointsException' needs to be included. For the centre case, more points are added to the card than are needed, and 'usePoints' is called within a try-catch block in case an exception is thrown. The points on the card are recorded before and after the points are used, and compared with one another to ensure the correct number of points have been deducted. Then, the edge cases of negative points and zero points trying to be used are tested in much the same way, throwing an exception if necessary. Finally, the important edge case of attempting to use more points than are stored on the card is tested, with the throwing of an 'InsufficientPointsException' leading to passing the test.

'getNumberOfUses' used one of the advantages of JUnit testing in using loops to make tests more rigorous. Here, I used a nested loop to call 'addPoints' and 'usePoints' several times while incrementing a counter for how many times each had been called. This allowed me to use the counter as a parameter in 'assertEquals' and get a reliable result for a test involving many operations. I also tested the edge case of an initialised card having not been used yet, and confirmed that the recorded number of uses was zero.

LoyaltyCardOwner

This class is very easy to test, as it only has two 'getters' for attributes that must be defined during instantiation. Both tests simply involve comparing the values passed into the class's constructor with the values recorded as attributes in the class and assuring that they are equivalent.

LoyaltyCardOperator

This is the most complex class of the three to test, as it has the highest number of and most complex methods. It also must contain data structures for recording all of the cards and their owners to allow for processing of certain methods.

'registerOwner' has a simple centre case to test of a valid owner being passed into the method, and provided an exception is not thrown, the owner can be considered registered. This test does not involve a 'check' of the data structure to ensure that the 'owner' object is present as the structure should be private to the operator class to ensure that owner's data cannot be accessed through other classes. The confirmation of the existence of the owner in the data structure can be 'checked' later when other operations call upon the 'owner' object for information. The edge cases of this method are attempting to register the same owner object, a new owner object with the same email, and a new owner object with the same name. For the first two of these edge cases, an 'ownerAlreadyRegisteredException' should be thrown, as the email should be unique for each owner. The email should be unique as each owner needs a single, unique, identifier, and the name of the owner is not reliably unique. However for the final edge case, two owners could have the same name, and thus the new owner should still be registered.

'unregisterOwner' is tested similarly to its registering counterpart. Again the data structure cannot be checked for data so the centre case test simply shows that no exceptions are thrown, which suggests that the operation executed as expected. The edge case of unregistering an owner that has not previously been registered should throw an 'OwnerNotRegisteredException' and again is encouraging in showing that the 'register' and 'unregister' operations are functioning as intended.

'getNumberOfPoints' is the tool needed to confirm the registration of an owner, as it can access the data structure to retrieve information about the owner. If the owner has not registered successfully, this information will be wrong or not exist. The test for the edge case of an unregistered owner's points helps to demonstrate this point, by throwing an exception if the owner is not registered in the data structure. The edge case of a registered owner not having made any points transactions is also tested, and confirmed to be zero. The centre case must be tested in conjunction with the process of money purchases, as this is the process that will be used to alter the number of points on a loyalty card in practical applications.

'processMoneyPurchase' has tests that thus build on the functionality of both registering an owner and retrieving the number of points associated with the owner's card. However, if all methods are operating correctly as is suggested by their own tests, the test for processing a monetary purchase is also specific and useful. A sample purchase is made, and the number of points retrieved from the owner's card stored in the operator's data structure is compared with the expected number of points to be added for such a purchase. The edge case of an attempted purchase for an unregistered owner is also included, and throws a relevant exception.

'processPointsPurchase' is tested in a similar way to money purchases, only with a money purchase itself made beforehand to represent points being added to the loyalty card before a purchase is attempted using the card's points. This also provides the obvious edge case of the loyalty card not having enough points to make a purchase, which is tested by ensuring the catching of the provided 'InsufficientPointsException'. As with money purchases, the case where an unregistered owner attempts to make a points purchase is considered and managed.

'getNumberOfCustomers' is simply tested by registering some owners, and asserting that the recorded number of registered owners is the same as the number of owners that were registered during the test. The edge case of no owners being registered and the recorded number of owners being zero is also tested.

'getTotalPoints' is tested with the centre case of multiple cards being used (both earning and spending points), and assuring that the total number of points in the 'system' is equal to the number stored on all of the cards both before and after the operations. In the edge cases of no purchases having been made or of no owners or loyalty cards being stored the method should return zero and not throw any exceptions as nothing program breaking has occurred.

'getNumberOfUses' has a centre case that is tested once but with multiple transactions, in each case, the transactions are successful, and the program should record an additional successful use of the card. With the edge case of a points purchase being unsuccessful due to insufficient funds, the number of uses should not increase, this is checked after the InsufficientPointsException has been caught to ensure a thorough test. Finally, when a card has not been used, the number of uses returned by the method should be zero.

'getMostUsed' has a central case for which multiple cards have been used, but one more than all of the others. The method should return this one card, and not throw any exceptions. In the case where two cards have an equal number of uses, the owner that was registered first is returned. I made this choice because it is a simple and reliable 'tiebreaker'. Finally, in the case where no cards are registered, null should be returned, and this is asserted in the final test.

Implementation

Here I will detail some of the choices and decisions that I made while ensuring that all of the tests were passed.
For loyalty cards, I decided to add an additional message to the 'InsufficientPointsException' to detail when an invalid number of points had been input to spend, such as a negative number. This would help the user of the system understand what the issue was, as the issue is distinct from the loyalty card not having enough points to make a purchase.
I used a Linked Hash Map as the data structure to store the owners, their cards, and the number of card uses in the 'Operator' class, as a map makes it easy to update data for a certain 'index' or key, which allows for a card to be updated easily without need to perform any processes on its owner. I chose a Linked Map to ensure that the order of owners was preserved when an iterator was used, which helped make the 'tiebreaker' for 'mostUsed' simple.
I also decided not to use the 'OwnerNotRegisteredException' for the 'getMostUsed' method, as my decision to use an iterator over the already registered owners avoids the chance that an owner will not be registered.

Conclusion

In conclusion, I think that I have successfully used a TDD process to create my implementation of this practical to suit the brief. I think that I have not made my tests completely atomic, but this is difficult to do with my self-imposed lack of access to the data structure in the 'Operator' class. However, I think that this access is important to restrict when the practical implementation of this program is considered. I feel that I have been thorough, but that in the process of development I have been worried about bugs in my test code as much as bugs in my implementation code, and that using a TDD style has introduced a significant amount of extra debugging. I do feel that working in this manner has helped me to learn to work to an interface, and justify my coding decisions effectively.