Overview

The second practical required us to develop an implementation of the ADT interfaces for a shop, stock records and products following a TDD process and therefore our job would be to implement all the given classes and write suitable tests. In order to do this, we would have to write JUnit tests for each element of functionality before writing and testing its implementation. The outcome of this practical would be to gain experience with Test Driven Development and implementing and testing Abstract Data Types.

Design – Basic

The framework for this practical was already given to us and therefore all we had to do was implement it and fill various method bodies. I have not edited the Exception, Interface and Factory classes.

The main part that we had to implement was the shop class. In the shop class I used two lists "products" and "stock records" which contain all the registered products and stock records respectively. The reason I used lists is because they do not have a fixed size and have various methods such as contains. The shop class is the main class and contains all the methods which give the application all of its functionality. The register and unregister methods allow new products to be added and removed from the products list. Furthermore, the "registerProduct" method automatically creates a stockRecord object and add it to the stockRecords list, likewise the "unregisterProduct" method automatically removes the stockRecord for the stock records list. If an attempt is made to register a product and there is already a product in the products list with the same bar code an "BarCodeAlreadyInUseException" is throw. If an attempt is made to un-register a product but the product doesn't exist in the products list an "ProductNotRegisteredException" is throw. The next two methods "addStock" and "buyProduct" are very similar to eachother, they both take in the barcode of the product and then find the stockRecord for that product and update the attributes for the stock record appropriately. Both methods can throw a "ProductNotRegisteredException" exception when a barcode is given in which does not match the barcode of any products. However, the "buyProduct" method can also throw a "StockUnavailableException" when there the stockCount attribute in the stockrecord object, relating to the product, is equal to 0. The other methods in the Shop class, preform operations on the products and stockRecords lists which return statics, such as the product with the most sales and total stock count.

I have decided to create individual classes for each of the classes un the "impl" folder, I also have one main test class called "Tests" this class when run, runs all the tests in all of the other test classes. In the test classes I have tried to include tests which test: normal cases, with expected inputs, corner cases, such as empty collections, duplicate bar codes for different products, no stock and exceptional cases, such as dealing with nulls. I have also tested every method in every class and attempted to test every branch where possible.
I have made use of the "@Before" annotation in order to reset the objects before each test. In order to test the methods which, throw exceptions, there are methods in the test classes which expect the exception to be thrown, I've done this by using the "expected=SOME EXCEPTION" annotation.

Design – Extension

In order to use the practical a database.properties file must be created

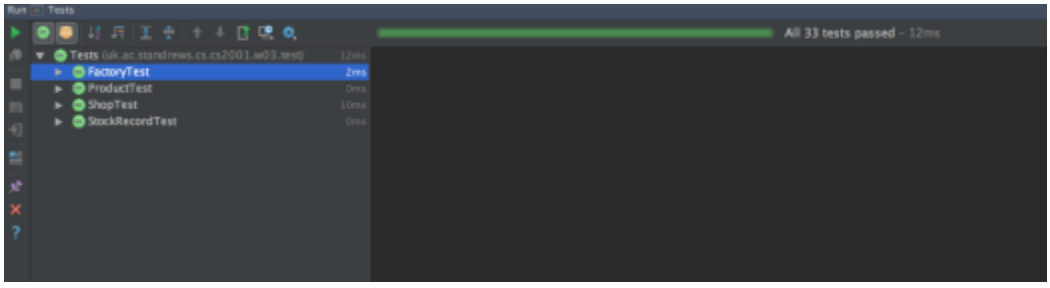
I have decided to implement an ORM system which now means that data can be permanently stored in a database as opposed to being transient data. I also wanted to use Junit to try test all the Data Access Objects I created.
In order to implement the system first of all created a "properties" class which reads in the database.properties file and sets up the connection to the database. I have also created a "Customer", "CustomerOrder" and "OrderLine" classes. Thus allowing the application to make and record orders which contain products and are made by customers. The "OrderLine" is a linking table between "CustomerOrder" and "Products". This was necessary because many customers in theory could order many products causing a many to many relationships. The other tables include a "Products" and "StockRecords" table. The "StockRecords" table has a one to one relationship with the products table. Meaning one product can have stock record and vice versa.

I have had to change some of the interfaces in the original interface, the main change was now the products have a "productID" attribute which essentially replaces the barcode attribute. This was necersarry because the "barcode" attribute was a string and thus could not be used as a primary key. I have also now edited the "addStock" method which now has an integer parameter "amount" which means the "stockCount" attribute can more efficiently.

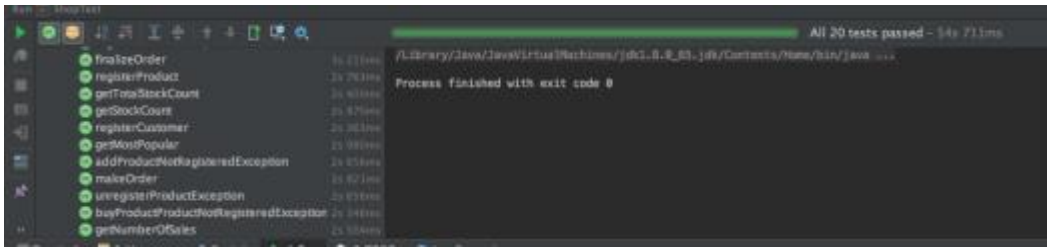
The shop class still acts as the HCI for the program and many of the methods still preform the same operation. However, I have now used an "EntityManager" object which allows the methods to store and retrieve data in the database. Moreover, the "EntityManager" has an "createQuerry" method allowing the program to preform queries on the data and manipulate it accordingly. I have added four new methods to the Shop class, these include: "registerCustomer" and "unRegisterCustomer" which work in a very similar fashion to the "registerProduct" and "unRegisterProduct" accordingly and the next method is "makeOrder" which records which products have been purchased and what order the purchased products belong to, and finally the "finalizeOrder" which records the customer whom made the order and the order date.

Testing
After using the automated tester all my 3 tests passed in my basic requirement, however due to the nature of my extension I have been unable to run the automated tests on it.
For my basic requirement I have written 33 different tests, all of which pass. I have then written a further 20 tests for my extension, again all of which pass.
I have decided on my extension not write tests for any other class's other than the shop class. This is because many of the tests would be repeated in the basic requirement or would be simply testing getters and setters.

Basic



Extension



Evaluation

After testing my program, I pleased to say both the basic requirements and the extensions it works as expected. I have used multiple classes and universal methods which take in various parameters making the program overall more modular and adaptable.

Conclusion

In conclusion I'm satisfied with end program, it is able to carry out the required tasks with ease and is able to handle erroneous data without crashing. Moreover, the use of ORM has made my program is to a usable application.