COM1028 Software Engineering

Software Engineering Report

Nithesh Koneswaran
6474079, nk00374@surrey.ac.uk

1. Introduction

This report provides a reflection of the design and requirements testing for my 'Online Market' name. It contains an evaluation of each of those phases and the experience gained through the project.

2. Reflection on Design

The graphical user interface of the project turned out to be great. However, there are still room for improvements. In the customer access level, the forms are separate JFrames so it pops up when the user moves to a new form. I believe this isn't very aesthetic and can be quite 'annoying' for the user, as having multiple forms can get quite messy and unorganised. This could have been greatly improved if all the features of the program were implemented on different tabs instead of separate JFrames. The use of tabs can help the user navigate through the application with ease. This is exemplified with my Admin Control Centre form. This form has a simple design, having a tab layout to allow the admin to navigate through users, products and admin activities. Other than that, I have no other complaints with the look of the software especially with the project's time constriction in consideration.

The project was quite time consuming, it was easy to deviate and work on features that were optional and not mandatory, this lead to time waste and code inefficiency. If I was to start the project again I would have taken a stricter agile development approach, defining sprints on a weekly basis and completing a set tasks within the specified time frame. Any tasks I had not completed would be written down into a backlog and will be completed at a later stage.

When displaying the list of products on the GUI form, this seemed to be quite procedural. I could have implemented an Iterator pattern that can iterate through all the products in the list and display them on to the JList making the project more Object orientated rather than procedural.

The main design pattern I used was a singleton class called CurrentSession. The class would have two fields a customer and an admin. One of these fields would be initialised and set depending on which type of user had logged in. Once one of the fields is set the program will determine the user's level of access and will proceed to the next form (Admin Main Form for admin/ Main form for Customers). Using a singleton class to keep track of the user who had logged in is something I would not change in my program. However, I believe that there should be a much easier way to set up the singleton class appropriately to keep track of the user that had logged in.

In terms of the coding, overall, I was not happy. I was struggling on how to use classes within the actual GUI and how it can interact with the database. I also instantiated objects strangely differently. I decided to instantiate the object using a unique ID, the unique ID will be passed into an SQL statement (in the constructor) that will retrieve the attributes and fill the object fields. I had believed this to be an innovative idea and had proceeded with this unique design pattern, although soon realised that using this too much can use up a lot of computer resources especially when looping through a list of ID's. Though my mindset upon using the paradigm was that I found it to be quite useful since storing arrays of objects could take a lot of space in the memory, therefore by storing only the ID of the object I would be saving a lot more memory as I could simply feed the ID into a constructor and then have access to all the fields of that object.

Another example I was not happy with was that in the mutators of my classes I have added SQL code so that whenever there is a change to the class's field through a mutator method, SQL code will then be processed to update the database. I believe that this way of linking the system model with the user interface and

database is inefficient. I should have applied abstraction and should have appropriately used a Model-View-Controller. By defining controllers, I could then respond to user actions which will then update the database in an abstract, organised manner.
To conclude I have managed to incorporate object-oriented programming into the project with barely any procedural programming. This makes the code easier for testing and maintenance. However, I believe that the way I have implemented is unstable and should have been done differently. I could have taken the design pattern of an MVC model to take advantage of the abstraction involved. The overall difficulty was quite hard, trying to implement a design pattern in a huge system proved to be quite difficult, if I was to reattempt this project I wish to further develop my knowledge and understanding of these design patterns so that I can correctly implement them and be satisfied with the end result.

3. Reflection on Requirements Testing

The requirement testing was very useful. I have only tested the mandatory requirements because it was important for the core functionality of the project to be working and finished. The requirement testing was very effective in finding and locating errors in my code, I had an error during test 8 , when removing a product the list of registered products remained the same. This error was only found due to the requirement testing. This demonstrates that it is important to test each functionality and step of the project to prevent small bugs and errors just like the one I had in test 8. Also, by going through the requirements I was able to look back at the code and make and quick adjustments that would improve the efficiency and maintenance of the code. I believe that the mandatory requirements have been tested appropriately, although there was not enough testing for the optional requirements. If there was more time I would have like to fully test and documented the optional features. I would also like to implement more scenarios in the test cases for some mandatory test cases such as 13, 14,16,17. This is because I could test the case when adding an item into a blank list or test the case when removing an item that leads to a blank list. I am suggesting that more precise scenarios will test the code thoroughly and allow me to reinforce the code to prevent any small bugs and errors.
I split my program into three main parts, the Login System, Customer access level and the Admin access level. I began working on the GUI taking it in turn for each parts of the program. After creating the GUI, I began to code the login system. After completing the login system, I started testing it against my functional requirements. After I was satisfied with the results I moved on coding the customer access level and so on.
There were scenarios during testing where I would have had to go into the database and manually add data. When testing the login system, I had to populate the user table manually and tested if the user was able to log in through the GUI. Similarly, when testing if a list of products gets displayed I would have had to go into the database and manually populate the Product table with products so that a list of products will get displayed. This was also the case when testing if a user could log in when they are banned or changing product's type from verified to unverified etc. I have iterated this technique throughout the implementation of the project helping me debug any errors, creating different scenarios to test and finally helping me create concrete code.
I have also included some Junit testing into my project. Junit worked well since my code was quite modular. If I wanted to further test the methods I could have added more comprehensive Junit testing to all the classes, though I was quite satisfied with the requirement testing. In my Junit test classes I had needed to add temporary data into the database so that the JUnit test methods can work as required. An example of this is adding a customer into the user and customer table. After adding the data I can then test the customer class methods on the customer that has just been added to the database. After the Junit tests have ran, the Junit class will then delete the data that was added to the database. Working with Junit had further allowed me to throw IllegalArgumentExceptions and NullPointerExceptions when needed and allowed me to pinpoint any errors in the code.