**Critiques**

* When comparing the artefacts in our analysis stage to what we had with our result, we noticed that the layout of the project remained the same. As for the longevity of our project, the concept we have provided will become inefficient as we continue to scale further up. The reason behind this is that using if/else statements in finding if group statements match would be considered poor practice for a major company’s website. As well, in the observer design pattern, the action in which rewriting the entire text file should the user change a discount or price on a product is fine for now, but should the file contain a few thousand products, we should eventually move over to a proper SQL database as it would be able to handle this type of volume.
* During our first iteration, we decided on the concept of hard-coding in our products and customers into the system. This idea was not efficient due to the fact that in later iterations, we would have the idea of adding more products or customers to the system, making this concept difficult to implement. We created separate text files to store all the information in regards to orders, customers, etc. This implementation of the text file databases lead to our project being more scalable and reliable come our end result.
* Originally, we were incorporating all of our classes into one default package. We were then made clear of that we needed to separate our system models from our UI classes. After going over the MVC approach, we then separated our system into 3 separate packages, each demonstrating the concept of the Model, View and Control layers respectively. The separation of responsibilities allows more flexibility with our code in later iterations and easier to then implement our design patterns into the system.
* One critique that we had for the Composite design pattern is that the Product class that inherits from component is present and is not even abstract, as it has no reason to be instantiated. However, throughout the code the many Objects of type product were being created by the time we realised the error. Instead of refactoring a lot of the work we had done (which would have been disastrous at this late stage) we decided to accept it as it was. Although it does not follow the typical Composite design pattern it works just the same – just with one unnecessary class in between.
* Early on in our iteration, we developed the concept that the user would be permitted to have the choice of either buying individual products or buying packages of computer systems that already included the individual products. This idea seemed too complex to implement in our code, and with time permitting, we decided to go with the alternate. Currently, we now have the option for the user to personally create their own PC where the system requires certain types of subclasses chosen from the database (see implementation of factory class). This change was beneficial towards our project as we were then able to implement the other design patterns into the system.
* Later on in the iterations, we included the concept of designing PCs and laptops. When purchasing the pieces to create these machines, the system requires certain subclasses in order create them. We thought of assigning each individual piece in the database their respected value, but this method would be too time consuming and complicated as the admin has the option of creating new products to be stored in the database. The Factory Design pattern was then implemented into the code to help of the assignment of the subclasses in the database. Using this design pattern demonstrated high cohesion and took away an unneeded tasks happening in the DataLayer class.