CSC 207 Phase 2 Design Document

Major design decisions:

Our team began with deciding the formate of our deisn: either a Android app or decktop application with GUI. After the first group discussion, we found that we were more interested in the second formate so that we deceided to develop a decktop application with GUI. During the development, our group could not deciede if our program should have multiple inventories or just one due to the incompletion of CRC model. Moreover, if our design could not provide an optimizing logic to improve the delivery speed for orders from different places, the multiple inventory approach was meaningless. Thus, after phase 0, we got rid of the idea of having multiple inventories. In phase 0, we also allowed User class to have two subclasses which were Admin and Customer. Sepecifictly, admins are in charge of the back-stage management, while customers are users who use our application to place orders containing require products in the inventory.

In phase 1, our group passed the decision about refactoring our UseCases into one specific Usecase folder in each packages for increasing packaging clarity. In addition, we used to store user's information, order history and item name and item quantity in corresponding entity classes. However, by combing feedback from TA, group members were responsible for the User, Order and Item packages choose to use local CSV files to store data instead of abstract classes in phase 2 for fulfilling data persistence.

SOLID:

SOLID design principles contain five class-design principles, which are relevant to reducing code rot and improving the value, function, and maintainability of software.

Single responsibility principle:

In our program, we've split the responsibilities of each use case in such a way that every class handles only one.

The SendMailUseCase class is only responsible for the action of sending email from our application, including the preparation and the exact process of sending mails. There are no other functions that the class is in charge of. Therefore, the SendMailUseCase is a detailed exmpale shows that our program adheres to the single responsibility principle

Since each class has a single responsibility we are able to change the classes freely without worrying about the effects of the change on other classes.

Open-closed principle:

Throughout programming, we have adhered to the open/closed principle closely, and this principle has allowed us to expand our program freely.

One example in our program showing the maintenance of the open-closed principle is that when we want to add another user type, such as supplier, in the future, we can simply create a new supplier class as the subclass of User class, which is an abstract base class, without changing any source code.

Liskov substitution principle

This principle is mainly applied to inheritance hierarchies. In our program, the customer subclass and the admin subclass operate in the same manner as their base class which is the User class serving as evidence of adhering to the Liskov substitution principle.

Interface segregation principle

This principle states that clients should not be forced to depend upon interface members they do not use. We did not implement any interface since no base class needs to be overrided more than once. In other words, abstract classes in our deisgn are all used once. In addtion, no classes in our program are forced to implement unnecessary methods.

Dependency inversion principle

This principle states that high-level modules should depend upon abstractions rather than low-level modules. In our program, the UI which has a function of sending emails is the high-level class and sendMessageUseCase is the low-level class. In our program, the sendMessageUseCase implements the detail of the function, which is called by the Message controller. Then, the MailUI calls corresponding method provided in the Message controller, hence UI and Use Cases depend on the controller rather than each other.

Clean Architecture:

Scenario walk- through:

For our scenario walkthrough, we will step through a standard scenario where our customer wants to create an account, sign in, place a new order, and receive a receipt. In terms of registration, the new customer presses the "register" button leading to the creation of the new registration UI. After filling username, password, email, and secrete code in the corresponding text fields, the register method in the UserController is called by pressing the "register" button and to verify if the entered secrete code matches the required one to be an admin. On the other hand, if they do not match, the user is identified as a customer. Then, the inputs are forword to the RegeisterUseCase Class and waited there to be stored in the local CSV file. After all the above finish, the local CSV file calls the writeUsers and readUsers methods in the Gateway Class for looping over the CSV file and encoding information.

To sign in, our program starts with the GUI which creates an empty login panel. Username and passwords are filled in the username text field and password text field by the new customer. The secret code is required for double checking if the user is an admin. After pressing the "login" button, the J button method then forward inputs to the gateway class in the User package. Then, the readUsers methods in UserReadWriter class and log_inUseCase class which are Use Cases are futhur called for comparing and encoding data. If the entered username and password matched the stored ones, then a pop-up window containing "login successfully" can be viewed by users. On the other hand, if the entered username and password failed to match the stored ones, another pop-up window saying "login failed" appears.

After signing in, the customer enters a new window and presses the “new order” button switching to the new order Panel. By entering wanted items in the text field, the item information is forwarded to the OrderGateway class for further processing. In the OrderController class, the method named Generate_Order_in_GUI is called which futhur calls Generate_Order_in_GUI method in the Use Case, while the OrderGateway calles file_writer method in the file_writer Use Case. Then Generate_Order_in_GUI method uses file_writer to encode information into the local CSV file. In the OrderGenerateUseCase, Generate_order_customer method creates the order entity which constructs an order object. After placing the order, Order Infomration has been stored in the local CSV file.

In the main panel, by clicking the “Message” button, a new MailGUI is created. The user can write content of the mail, subject and receiver account into the text field. The input of the user will be passed to the MessageController. Then the method of sending email in the MessageController will be called, which will lead to the calling of send_mail_admin/customer method in the SendMailUseCase. After that, the message that indicates the success of sending the email will be generated and presented by the MessagePresenter to the GUI.

Since the scenario walk-through covers the functionality of our code partially, you can move to the CRC model to view the full structure of our code which follows the clean architecture strickly. The following is the link to the CRC model of our group:
https://miro.com/app/board/o9J_lq5DFgs=/?utm_source=notification&utm_medium=email&utm_campaign=daily-updates&utm_content=go-to-board

Packaging strategies:

The package strategy that we used was strcturing layers by components so that all of the data and functions inside each component are semantically related. For example, the message package including the controller and presenter and Usecases are responsible for mail functions merely.

Refactoring:

Three notifications containing instructions and a to-do list have been created by Gen (Reagan) LI through the Issue feature of GitHub. Each pull request and workflow run has a clear and specific title, so it’s easy to figure out what each of us has done while looking through them. Moreover, keeping everyone in the group on the same progress can be accomplished effortlessly by defining changes through these features of GitHub. The following two pull requsts demonstrate that our code had been refactered in a meaningful way.
Pull request #112: https://github.com/CSC207-UofT/course-project-bug-producer-1/pull/112
Pull request #118: https://github.com/CSC207-UofT/course-project-bug-producer-1/pull/118

Use of GitHub Features:

Meilun Shen: https://github.com/CSC207-UofT/course-project-bug-producer-1/pull/65

In pull request #65, Meilun Shen re-structured the whole User package for following clean architecture. In order to fulfill data presitence in phase 2, she added the database gateway and Bufferreader and Bufferwriter methods for accessing the CSV file for storing user information.

Gen Li: https://github.com/CSC207-UofT/course-project-bug-producer-1/pull/53
In terms of the pull request #53, Gen Li (Reagan) proposed the changes about refactoring the structure of the controller, presenter and specific codes in the UseCase classes. This restructuring would implement the messaging system function and refactor the code to follow clean architecture. Because the Message Controller is responsible for take requests and inputs from the GUI/users and call UseCases to do the action which is needed. After that, the result generated by the calling of UseCase is caught and processed by the Message presenter. Finally the Messgae presenter generates and presents the information of the processed result to the UI/users.

Michelle: https://github.com/CSC207-UofT/course-project-bug-producer-1/pull/70
Michelle has been working on the database and its gateway for items. Two methods were written in the database in order to read and write a csv file containing the item names and capacities. The above is the link of a significant pull request. It is significant because Michelle has modified the class using an easier storing method such that we can read and write the csv file, while previously the class was more complicated.

Xingru Ren: https://github.com/CSC207-UofT/course-project-bug-producer-1/pull/4#issue-1024549533
Xingru finished whole Item part by this pull request. And this class is one of the entities class for our project.

YueHao Huang: https://github.com/CSC207-UofT/course-project-bug-producer-1/pull/2
This pull request explains what YueHao has been done and what needs to be done in the near future clearly and has tag properly set.

Chongjie Sun: https://github.com/CSC207-UofT/course-project-bug-producer-1/pull/119
This pull request improve the mistakes in inventory, making inventory easier to use. At the same time, after this push, it successfully cooperated with the gui, so that the entire project can run normally.

Hao Li: https://github.com/CSC207-UofT/course-project-bug-producer-1/pull/68
Hao Li thinks this was a big milestone for our project. Through the communication with my teammate who was working on GUI. We were able to nearly perfectly run one of our core features crating orders and order history. In addition, we were able to write usable methods and reduce workload for both through proper communication.

Design Pattern:

Since the message part is mainly responsible for the communication between the UI and core codes, having a controller in the Interface Adapter layer for the calling of functions, which are sending emails, creating message entity and a new varible of DefaultlistModel, in core program codes is important. The MessageController provides these three methods, so that GUI can simple call them without accessing to the complex core program codes. The above is one exmaple out of many in our deisgn showing employment of the Façade Pattern. The Façade Pattern simplifis steps for higher-level layers accessing the functions in the lower-level layers. Moreover, By using the Façade Pattern, the safty of the program increases due to the inaccessibility of the core level codes to users.

In the presentation, Chong jieSun mentioned that inventory uses a design pattern called state. But afterwards, he chose to delete this design pattern. At first, he found it useful because it can clearly show the internal state of inventory. But afterwards, we discussed together in the meeting and reached a conclusion. Because our project does not need to display the status, we only need to know the number of items stored in the inventory. Compared with intuitive numbers, a state is more vague. This shows that the design pattern does not play the role it should have, so that we chose to delete this design pattern.

Adressing any IntellJ Warning:

In the User Package, there are a few weak warnings, mainly are inspected in the GetCurrentUser Class, indicating unused functions. Meilun Shen, who implemented the whole User Package, did not delet this class because it serves as the backup plan for getting current user's information. Right now, the approach of getting current user's information in our deisgn is through saving this information in a constant class locates at the outset layer. In case this approach fails in the future, the GetCurrentUser Class is a alternative solution and port for GUI to resolve this problem. Therefore, we decided to keep this class in our program.

The following link is the link to the GetCurrentUser Class: https://github.com/CSC207-UofT/course-project-bug-producer-1/blob/main/src/main/java/user/useCase/GetCurrentUser.java