CSC 207 Phase 2 Progress Report

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. The first principle is the single responsibility principle which states that there should never be more than one reason for changing a class. In other words, every class should have only one job to do. The following figure shows that our design adheres to the single responsibility principle.

文本

描述已自动生成

Figure 1

Here we have one class only contains one method which is responsible for defining the context of the message for Admin. The open-closed principle, which is the second principle in SOLID, states that classes should open for extension and closed for modification. 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. Then, the Liskov substitution principle applies 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. The interface segregation 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 has unique functions in our program. The dependency inversion principle, which is the last 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, this UI calls corresponding method provided in the Message controller, since UI and Use Cases depend on the controller.

Clean Architecture

In our CRC, we, mainly Reagan, group entities, Use Cases, and controllers based on different packages. All GUI classes and the Database belong to the outer layer of clean architecture, also known as frameworks and drivers.

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 through email. In terms of registration, the new customer presses the “register” button leading to the creation of the new registration GUI. After filling username, password, email, and secrete code in the corresponding text fields, the register method in the User Controller is called by pressing the “register” button. Then, the register method in the register Use Case is called for verifying that the entered secrete code matches the saved one. If they do not match, the user is identified as a customer. Next, the above information is stored in the database by using the register method in the register Use Case.

To sign in, our program starts with the GUI which creates an empty login GUI. Username and passwords are filled in the username text field and password text field by the new customer. After pressing the “login” button, the J button method then calls the log\_in method in the User Controller. And the log\_in method verifies if there is a matching between the username and password and the stored username and stored password in the database. If the log\_in method returns the value of “true”, then the current log\_in GUI is disposed and a new main GUI is created.

After signing in, the customer enters a new window and presses the “new order” button switching to the new order Panel. By selecting and entering wanted items in the wish list text field, the item\_list in the Inventory Controller is used to call the Add\_item method in the Add\_itemUseCase. Add\_item method compares inputs from wish list with item\_list attribute in the inventory entity. If Add\_item method returns fault, then the current wish list is canceled. If the add\_item method returns true, then the inputs from the wish list are forwarded to the OrderGenerateUseCase for further processing. In the OrderGenerateUseCase, Generate-order-customer method creates the order entity which constructs an order object.

After placing the order, a receipt is delivered automatically by using the send\_chat\_message\_mail method in the Message Controller Class.

One clear violation of clean architecture in our program is three Use Cases importing the Database. In other words, the inner layer references a layer above it. In addition, according to the explanation of the scenario walk-through, the interactions between UI and various Controllers demonstrate our program follows the dependency rule. However, as mentioned above, the maintenance of the dependency rule is failed within the user class. In phase 2, we will mainly focus on and revise this issue.

Design Patterns

So far, our group did not have enough time for implementing any design patterns. However, some design patterns can be used in our program and will be implemented in the future. The factory method design pattern can be used in the User package. We have a User abstract class and concrete classes (admin and customer) that extend the User abstract class. To implement the factory method, a factory class GetUserFactory should be defined. Admin class can use GetUserFactory to get a User object. It will pass information, such as admin, customer, to GetUserFactory to get the type of object it needs.

Use of GitHub Features

Till now, our group has created 34 pull requests with descriptions and over 140 workflow runs in the repository. Three notifications containing instructions and a to-do list have been created 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 page can be accomplished effortlessly by defining changes through these features of GitHub.

Graphical user interface, text, application, email

Description automatically generated

In terms of this specific pull request, Reagan proposed the changes about installing javamail and jaf for using the function of sending email. By reviewing the color-coded changes in the repo, other group members can discuss the potential changes with Reagan and add follow-up commits before Reagan changes are merged into the base branch.

Code Style and Documentation

Since all classes and methods come with polished Javadoc, we believe other Java programmers are able to understand our code, when they open our repo and navigate to a random file. Below is a screenshot for demonstrating the code style and documentation.

Graphical user interface, text, application, email

Description automatically generated

The Javadoc of MessageForCustomer Class states that the interaction between this class with other class, and its role within Message Package. Then, the Javadoc of the MessageForCustomer method explains what this method can do and demonstrates the context format with an example. Finally, four parameters of MessageForCustomer method are discussed in sentences.

Testing

Even though current tests can function and be used for testing our system, our program has not included any exceptions yet. For instance, so far, the minimal quality of items which is the parameter of stock\_in and stock\_out method in the inventory system has not been created. In other words, inventory can keep stocking out even if the capacity of items reaches 0. Furthermore, we have not excluded the situation when two usernames are identical. As the result, the information comes with the latest username will cover the previous ones. Another example of testing difficulty is within the Message package. The sendChatMessage\_mail method is a void method indicating no return value so that it is difficult for testing this feature. Since the time is tight for phase 1, we will definitely revise and implement these inaccuracies in phase 2.

Refactoring and Code Organization

Below are screenshots of specific pull requests relevant to refactoring in a meaningful way:

Graphical user interface, application

Description automatically generatedGraphical user interface, text, application, email

Description automatically generated

The main principle we followed for refactoring and organizing code is the clean architecture. We grouped code based on their entity at the beginning. Then each of us created and named new classes within the package as needed during implementation. Last, we gathered use cases within the same package to a folder for further clarity.

Based on what we have learn during lecture, we think that there are not obvious code smell existing in our program; otherwise, we failed to identify them.

There is one obvious code smell existing in our program: long parameter list code smell which appears in the RegisterUseCase and UserController. We have 5 parameters in the “register” method, which can improve in the future by creating another class to store these parameters.

Functionality

Our program functionality matches the specification. And Walter who devoted to implement GUI will present a live demonstration during the presentation on Nov.15.2021.

In the future

1. Polishing code by implementing design patterns, revising code smell, and restoring clean architecture, etc.
2. Adding shopping cart feature.
3. Using another deploy cloud-native application to store data (username, password, and other).

In phase 1, the individual contribution is shown in the following chart:

|  |  |  |
| --- | --- | --- |
| Individual Contribution | | |
| Meilun (Elva) Shen | Implementing User Package and tests; revising Specification; Writing the progress report | CRC model revision  Communicating and helping each other with the interactions between different packages. |
| Gen (Reagan) Li | Implementing Message and User Package and tests; revising the general structure of CRC model |
| Walter Huang | Implementing GUI and tests |
| Chongjie (Kurisu) Sun | Implementing Inventory Package and tests |
| Hao Li | Implementing Order Package and tests |
| Xingru (Jamie) Ren | Implementing Item Package and tests |
| Cheuk In (Michelle) Tam |