Conference Management Software Development

CSC207 Project Design Package

DECEMBER 11, 2020

Group 0014

Haohua Ji
Ziwei jia
Zhongyuan Liang
Yufei Wang
Jun Xing
Zhou Ye
Jiahao Zhang
Hanzhi Zhang

Table of Contents

Design Assumption:

Client: Toronto Convention Centre

Toronto Convention Centre has established relationship with all public universities in Ontario. As part of collaboration, Toronto Convention Centre is providing rooms for conference uses. And now the client is hiring our team to create an app to help them streamline this operation.

Four System Users:

- Organizer: Ontario's public universities
- Speaker: professors/professional individuals
- Attendee: students/public individuals
- Admin: people who manages the accounts of the software

Accounts and Login:

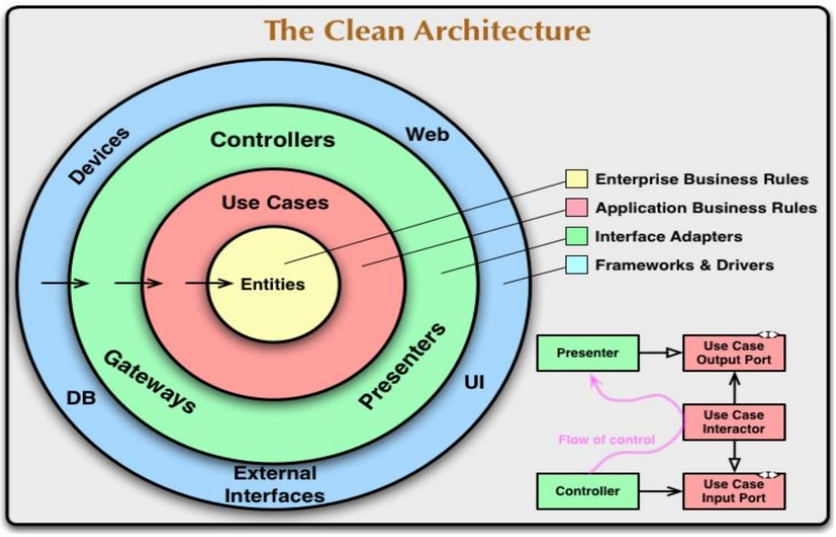
- A User account can only be created with valid(non-empty) email. And same email can only create one account.
- Only attendee can register, but all types of User can login from the user portal
- One Admin account, Organizer account, Speaker account have been written into database, please see details in the ReadMe file
- An Organizer can create accounts for attendee, speaker, and another organizer

List of features

- All features from phase 1 including register, login, signup for event, message and organizing events (see details in Readme)
- Mandatory extensions required for phase 2
- Optional extension:
  - Enhance the user's messaging experience by allowing them to "mark as unread", delete, or archive messages after reading them.
  - Database using SQLite
  - GUI using java Swing
- Own new features
  - Allow the admin to ban the user
  - Add "back" button and "logout" button
  - Allow the message sender to see available receivers so they do not have to memorize

Design Considerations:

Architecture:



This project follows clean architecture.
Seven package has been established, and they are Entity, Use Case, DAO, Controller, Presenter, Gateway and UI.
The relationships of all classes have been highlighted in the UML diagram.
We have 11 classes and 1 interface in the Entity layer.
The User and Room class implemented the schedulable interface.
Reason: Both user and room can be assigned to specific event based their availability at a specific time. And there could be other types of objects later on that can also be assigned to the event such as a TV, a projector. We can simply create these class and let them implement the schedule interface and it conforms to open/close principle.
User class has 4 child classes: Speaker, Attendee, Organizer and Admin
Event class has 3 child classes: MutilpleSpeakerEvent, NoSpeakerEvent, and OneSpeakerEvent. Although the only difference between the 3 classes is number of speakers, we encounter that there could be different methods between the three in the future, so we separated them into different classes.

In the second layer, we have the Data Access Object(DAO) classes and Use case classes.
Use case classes performs operations/service to entities based on business logic which were defined in the list of features.
DAO classes are meant to covert attributes stored in Entity classes to a format which most convenient for the gateway/database.
In the third layer, we have Gateway, Presenter and Controller.
The gateway class only depends on the DAO classes.
We did partial façade pattern in the loginfacade class which encapsulated UserAccountsController and LoginPresenter Class. As a result, The UI classes only depends on the loginFacade, MessageController, SchedulerController, and SignupController.

Solid Principle:

SRP:

- Classes are design to response to single actor. For example, UserAccountManager only handles tasks related UserAcounts/login/register. MessagerManager only handles tasks related sending/receiving messages.

OCP:

- we have the Schedulable for Room and Speaker so that ScheduleController can treat these objects as same thing when scheduling an event
- We have OneSpeakerEvent, MultipleSpeakerEvent and so on to extend from the Event class.
- All entity classes are independent from each other in our design
- All manager classes in Use case layer are independent from each other in our design

LSP:

- we have checked to make sure replacing parent class instance with its child class instance does not change it behavior. For example, We have a SchedulableManager that gives new schedule, check schedule to Schedulable no matter what classes are implementing the Schedulable interface

DIP:

- We have made classes to rely more on the abstract class. For example, The ScheduableManager only depends on Scheduable interface.

Design Pattern:

1) Iterator

We have implemented Iterator design pattern for class EventManager and RoomManager because these classes store a list of entity and accesses and modifies one of the entity from these collections. Most of the methods from this class takes an ID of one of the objects from the list stored within this class and need to loop the whole list to do some changes to that object. Therefore, we decide to implement the iterator design pattern so that when we loop the element within the list, the code can be more compact within the EventManager and RoomManager class.

2) Façade

We used Façade design pattern partially where the LoginFacade encapsulated the implementation of UserAccountsController and LoginPresenter. So the outer UI class only need to interact with this LoginFacade class for login/register function.

We used Façade design patter completely in the relationship between MainGateway class and the DAO classes. The UI classes only need to interact with the MainGateway class and do not care the implementations of the DAO classes.

3) Dependence Injection

Instead of creating new UserAccounterManager class in the UserAccountsController class, we used the dependence injection. We added AccountManger to the parameter list of the constructor of UserAccountsController and let the UserAccountsController reply the a interface instead.

4) Data Access Object(DAO)

Although we did not cover this design pattern in class, but the DAO classes allowed us to separate the business logic/operation classes from the data operation classes. As a result, in case we want to change database later on, we do not need to touch the business logic/operation classes.

Scheduling Implementation Detail:

When we schedule an event, we need to consider the time and the availability of Room object and Speaker object. Therefore, we introduce an interface Schedulable to abstract this property of Room object and Speaker Object, which seems unrelated to each other. In Schedulable Interface, we assume the class that implements this interface has some kinds of schedulelist which records the time of unavailability of this schedulable and we within the interface we have methods that deal with this schedulelist including check availability, add a new schedule, delete a schedule(setter of a field) and get a schedule(getter of the field). Then going to the use case, we have EventManager responsible for storing and modifying a single event in the system and a SchedulableManager specifically dealing with only the interface of Schedulable. This class only stores methods that takes the list of Schedulable in the system which is passed from RoomManager and UserAccountManager. Then the ScheduleController could use these two methods to schedule an event. The advantages of doing that is first we have a Schedulable interface which reflects the open/ closed principle. Since as the program develops, there will be many varieties of events, we may have more schedulable items for an event to take place. We wish we won't have to implement each of the methods for these classes like adding schedule, checking schedule for these classes again. Therefore, we introduce Schedulable Interface so that the program can be open to more entities as the program develops. Next, we have a SchedulableManager that deals with only the interface Schedulable. This reflects the two SOLID principle: Liskov substitution principle and Dependency Inversion. The methods of this class only depends on the interface and no matter the class that implements this interface is a person or an object, it can still do the same things to them without altering the correctness of the program.

Event Signup Implementation Detail:

Next, when the user wants to sign up the events, two classes are responsible for that: SignupManager and SignupController. The SignupManager uses a hashmap to store the correspondence of a single user to the list of events he signup. The controller will help the user signup and cancel the events.

Finally, notice each class in our use case is completely independent from each other. When we do want them to interact, we group them within the controller class. I believe this just reflects the Single-Responsibility principle of our class. Each Manager is only responsible for storing and modifying the certain entity classes. When the user wants to make an action, the controller will do the underlying things. Each layer only depends on the layer beneath them and this follows the clean architecture strictly.