CSC207 Group_0070

<u>Design patterns:</u> (By packaging)

user_system:

- Factory Pattern not implemented
    - We don't need to make object creation dependent on user type, and use abstraction to create methods based on one categorizing element, i.e. user type, since our phase1 design did not have different subclasses for different kinds of user.
    - Would only make sense if defying features for each user type was tied into the User System.
    - Enumeration is used in place.
- Observer pattern not implemented
    - For storing and accessing user information, since it is not changed throughout the program running (only one user logs into the program at a time), it was not feasible to implement an observer.
- Applied Dependency Injection at the controller level
    - We know that since the other systems may require access to the same user entities read by our user controller, such as the current user's login information, passing the UserService class inside not only this controller but other controllers as well allows us to have a consistent list of users that every controller can manipulate by calling methods in the UserService class.
    - Used in: UserService, UserController

event_system:

- Did not use the factory design pattern for different types of events
    - The only difference between the types of events is the number of speakers, no other difference is tiled into the Event system.
    - Similar to the design choice with different types of users, we don't need to make event entity creation dependent on event type; we use abstraction to create methods based on one categorizing element i.e. the enum event type.
- Applied dependency injection at the controller level for better encapsulation
    - Methods from other system's use case classes are required to be called in the controller, such as assigning valid, existing speakers (method calls for user system use case class) to an event when adding new events. To avoid instantiating new use case classes, the use case classes from other systems are being passed into the controller. This also avoids delay in updating information as all the use case classes are being shared.
    - Used in: EventService, EventController

messaging_system:

- Did not use Observer design pattern for read/unread

    - Observer pattern is in its optimal effect when there is an obvious cause-effect relationship and when there are multiple Observers for one Observable. In our design, we found it was unnecessary to implement the Observer pattern because when a message is marked READ/UNREAD, the only effect is that the message is printed to screen with its new READ/UNREAD status. There is no cause-effect relationship as changes in Message do not affect any other object.
    - We could argue that an Observer pattern might be beneficial if we want to include visual elements (like a red dot) in UI to indicate READ/UNREAD status instead of printing words "READ/UNREAD" onto screen in the future. However, the Observer pattern requires extending a concrete class, which is not as flexible as implementing an interface, and it adds complexity to our program class structure. Hence, we decided the Observer pattern was unnecessary.
- Did not use Builder design pattern for Message
    - Message objects do not require complicated, ordered steps to be built or sent.
- Applied dependency injection at the controller level for better encapsulation
    - Methods from other system's use case classes are required to be called in the controller, such as getting the list of attendees who sign up for an event to message all of them given a speaker name. To avoid instantiating new use case classes, the use case classes from other systems are being passed into the controller. This also avoids delay in updating information as all the use case classes are being shared.
        - Ex: a new event is created by the event use case class, and there is only one event use case class being instantiated, so messageController has access to the attendee list for that newly added event as the event use case class is being passed into the controller)
    - Used in: MessageService, MessageController

graphic_user_interface:
- Applied Builder Patterns to GUI classes.
    - Reason: There are a lot of parts of the GUI that need to be built sequentially, such as the separate components, the menu window and the login window.
    - MenuBuilder
        - Needs to build components sequentially to the primary stage of the window, which includes the separate components, the scene and setting up the stage.
    - LoginBuilder
        - Same as MenuBuilder but also needing to hide all the menu components when the login window is shown.
    - ComponentBuilder
        - There are multiple steps to loading a fxml component into the main class: Make a loader, get the file location, instantiate the model, and get the controller.

        - We thought it would be a good idea to encapsulate this idea into a class GUIComponent, and make a builder called ComponentBuilder because this helps with O/CP and single responsibility principle.

database:

- Did not use the factory/builder design pattern for different types of tables to create based on how each system's Gateway class stores the information
    - The only difference between the tables to be created in the database is a string of column names, the class variable is called sql_table. We are not creating any subclasses in the backend that share a similar method like factory, nor are we building complicated objects for a potential antipattern of builder.

<u>New Features:</u>

Extensions (4 of them)

- (1) Message_system - enhance the user's messaging experience by allowing them to "mark as unread", delete, or archive messages after reading them
- (1) Database - storage of information while incorporating clean architecture
- (2) GUI

<u>Design choices related to Clean Architecture, MVP/C, and SOLID</u>

GUI

- The implementation of a GUI has been made into a MVC pattern, using FXML as views to display the information, Controllers to pass information between the view and model, and the model which connects the controller to the backend.
- Views, (aka FXML files), belong in the UI part of Clean Architecture, as they are the medium which the user can see and interact with.
- Controllers belongs, (surprise), in the controller layer, they take input from the user via the views and passes that down to the model
- The model interacts with the entire backend of the system. Modifications to the information in the model will be displayed in the views.

Database:

Make every system Gateway classes a child class of a parent class that has methods to establish connection with database

- The parent class is abstract with read, write methods that are abstract; this enforces each child class (XDatabaseReadWriter) must implement read(), write() methods
- It is more extensible since we can have other kinds of Gateway classes (XDatabaseReadWriter) that integrate well with the database (create different kinds of tables in the database), and no dependency of storing a new class that deals with database operations into each gateway class.

Refactoring from Phase1 Feedbacks:

1. <u>(Feedback 1 and 3)</u> Consider your method names and class names for readability

   a. We have renamed the method names and classes that follow a similar naming convention, some examples are
      i. *userToPass* is changed to *validateCredentials*
      ii. *XAccountDataProvider* is changed to *XReadWriter*, then to *XDatabaseReadWriter* since we implemented a database
      iii. authorization/validation method names are changed to *isX* or *canX*
2. (Feedback 2) Check interface usage for classes that are likely to replace or for invert dependencies
   a. We have removed the gateway interfaces since they are only accessed in their own package (x_system) and there isn't any inverted dependencies for gateway classes
   b. Other interfaces are kept for invert dependencies and are accessed from other package/system, for a better abstraction
3. (Feedback 4-5) We maintained how we package the classes by theme, and classes in each layer follow the same naming convention across the theme package in addition to the GUI classes.
4. (Feedback 6) Go through some longer methods and extract helper methods, make them easier to read, replace nested if/else statements
   a. Helper methods are extracted whenever possible
   b. Suggested by our TA, we replace nested if/else statements (tree pattern) to a list of if/else statements (linear pattern) for readability
   c. The textUI in phase 1 had many nested if/else statements based on return values from the systemControllers (backend), the GUI classes replaced them with switch cases as the systemControllers return information (an enum info from XPrompt) related to why a action cannot be done, so the GUI can display the information for a better user experience
5. (Feedback 7 - 8) Remove MessageAuthorizer as it is very small, and put authorization checks into the Controller level as private methods
6. (Feedback 9) breaking up long functions like the ones in EventSchedulingSystem, which would make your code more readable
   a. We have replaced textUI with GUI (EventSchedulingSystem was for textUI), and long functions are extracted with helper functions or design patterns so it is more readable (check design patterns discussion above)

Others:

1. We have extracted out the nested controllers situation for phase1 as each systemController wants to get information about the current login user. We decided to make it as a parameter for methods to be passed into the controller instead.
   a. Following the idea of more O/CP to support more users to login into the program for future development, we overload methods with currently login user information to support single user login for this project, as well as maintain minimum changes to the test cases on backend (the time to change test cases were put into debugging frontend and look closely to codes for antipatterns to apply design patterns)
2. Use abstraction for builtin objects such as replacing ArrayList with List

3. Use enum (XIndex classes) for indexing of the String representation of entity to display at the UI layer, so outer layer do not need to know which array index corresponding to what information for a better encapsulation
4. Use enum instead of String for type of users/events to avoid having final static variables of methods, and a better readability & clarity of the backend codes

Thank you for a great semester and Merry Christmas :)