Group_0065 Design Patterns

Dependency Injection:

Class involved: UserSystem, SpeakerSystem, OrganizerSystem, AttendeeSystem, AdminSystem, EventSystem, AccountSystem, MessageSystem, Message, Event… (basically every class has a certain degree of dependency injection involved)

How and Why: We created the parameters for the above classes before we call the constructors and methods, and we pass in them as parameters instead of creating them inside. For example: When constructing a SpeakerSystem, we first construct username, eventSystem, messageSystem, roomSystem before the constructor call. Then, we pass these in as parameters.

Factory:

Class involved: AccountFactory, EventFactory, UserSystemFactory

How and Why: Since depending on which account we need to create needs to call different constructors for the specific account, we use an AccountFactory to decide which constructor to call. Similarly, when we need to create a certain kind of event and Usersystem we want to create, we use EventFactory and UserSystemFactory to decide which constructor to call.

Builder:

Class involved: UserSystemBuilder

How and Why: In our design, we have different types of subclass of user system, the construction of those different types of user systems require many steps such as reading from file and setting up lower level controllers like EventSystem, MessagingSystem and RoomSystem. Most of these steps are the same for different types of subclasses of the user system, since we want to make our code reusable, and we don't want to make our code too complicated, so we used the UserSystemBuilder class.

Strategy:

Class involved: InputStrategy, UserInput, RoomUserInput

How and Why: In our design, we are using Scanner in many controller classes to get user input, and some of the systems require different formats of user input. To ignore repeating the same code a lot of times, we decided to write the UserInput class to store methods that can decide if the input is in the format we want or not. However for different controller classes, they might have different format requirements. Hence, we used the InputStrategy interface to let all

different kinds of UserInput classes implement it. Those different UserInput classes will read numbers, times or strings for different requirements. And if we want to add some new features in the future which may have a new requirement for user input, the strategy class would help us to follow the Open Closed Principle.

Patterns that we did not include:

Iterator: In our design, we do have different managers that store a collection of accounts, events, etc. However, take AccountManager as an example, since we want to get different accounts by usernames and types, we use a map to store the accounts instead of a list, and didn't use the Iterator pattern.

Observer: In our design, the cause-and-effect relationship is hard to encapsulate into Observer Design Pattern. When a certain change is introduced by the user, each EventSystem, AccountSystem, MessageSystem and RoomSystem will react differently. Since there are many different changes that can be introduced by the user, and different system is going to react differently we fill like it's not necessary to use the Observer Pattern in our design.

Facade: Facade can be used when a single class is responsible to multiple "actors". In our design, UserSystem does contain references to EventSystem, MessageSystem and RoomSystem, where when needed UserSystem are just calling methods within them. However, since UserSystem does have its own functionality (like handling inputs from users, options for users...), we cannot say that our design implements the Facade Design Pattern.

Adapter: We didn't reuse anything from another program, we didn't use the Adapter pattern.