The design patterns used in our program

Design pattern: Observer

Name of all classes involved: Item, TradableUser

Why/how: For the “how” part, we made item class the observable, and TradableUser class the observer. We implemented this pattern with PropertyChangeListener. The reason why we implement this pattern is because we want to separate or encapsulate the cause (the change in item’s tradable status) from the effect (add a formatted string indicating the cause to an attribute in the TradableUser class). Even though this isn’t a case where the dependency shouldn’t occur as both classes are entities, it still brings better encapsulation and solves the problem of having to gather all the “effect” classes for the “cause” class to make an effect on. It also makes our program more open-closed as we can easily change the observer as well as the observable.

Design pattern: Dependency Injection

Name of all classes involved: all the GUI classes (except the notification GUIs), TradeManager, MeetingManager, ItemManager, SystemMessage

Why/how: To avoid having too many dependencies, instead of making the various controllers or presenters attributes of the GUI classes, we injected them in as parameters. By doing so, we are able to reduce coupling between these classes and loosen the dependencies. The construction details of the controllers or presenters are completely encapsulated from the GUI classes. It also makes our program more open-closed because we can easily delete, add, or swap the parameters. We also used this pattern for the use case classes and presenter class listed above to solve the problem of needing to call methods in classes of the same or different layer while still able to have good encapsulation, low coupling between classes, and also be open-closed.

Design pattern: Facade

Name of all classes involved: UserManager, UserInfoManager, UserItemManager, UserCommunityManager, UserThresholdManager, SystemMessage, PrintObjectMessage, TradeSystemMessage, UserAlertMessage,

AccountSystemMessage, MeetingSystemMessage, ThresholdSystemMessage, GeneralSystemMessage

Why/how: UserManager was getting too large and was becoming hard to navigate and read and difficult to test. By using the Facade design pattern the UserManager had less coupling with the classes that used it and also allows better encapsulation by not showing what the functions do. The original UserManager was split into 4 smaller managers each with different responsibilities related to Users and the UserManager called on the functions of the smaller classes whenever it was needed inside its methods.

The SystemMessage class also has the same problem. Not only it had over 500 lines but it was also hard to navigate. By using Facade design pattern and splitting SystemMessage class to the various other message classes (ones that come after SystemMessage in the above list), we're able to reduce the size of this presenter class, making it very easy to navigate, let it adhere better to the Single Responsibility Principle by splitting this class to subclasses with specific purpose and reduce the number of actors, allow the program to have better encapsulation with the same reason as the UserManager one and also allow the program to adhere to the open-closed principle better(ex. it'll be easy to add a new sub-SystemMessage class or delete a sub-SystemMessage class or swap one for the other with very little changes needed to be made).