Design Patterns

1. Observer Pattern

The Observer Pattern is typically used when one object (the subject) needs to notify other objects (the observers) about state changes, without knowing who or what those observers are. In the case of your system, the Tour class could be the subject, and Users or Admin could act as the observers that receive notifications when certain actions are taken (e.g., new feedback, updates to the tour).

Explanation:

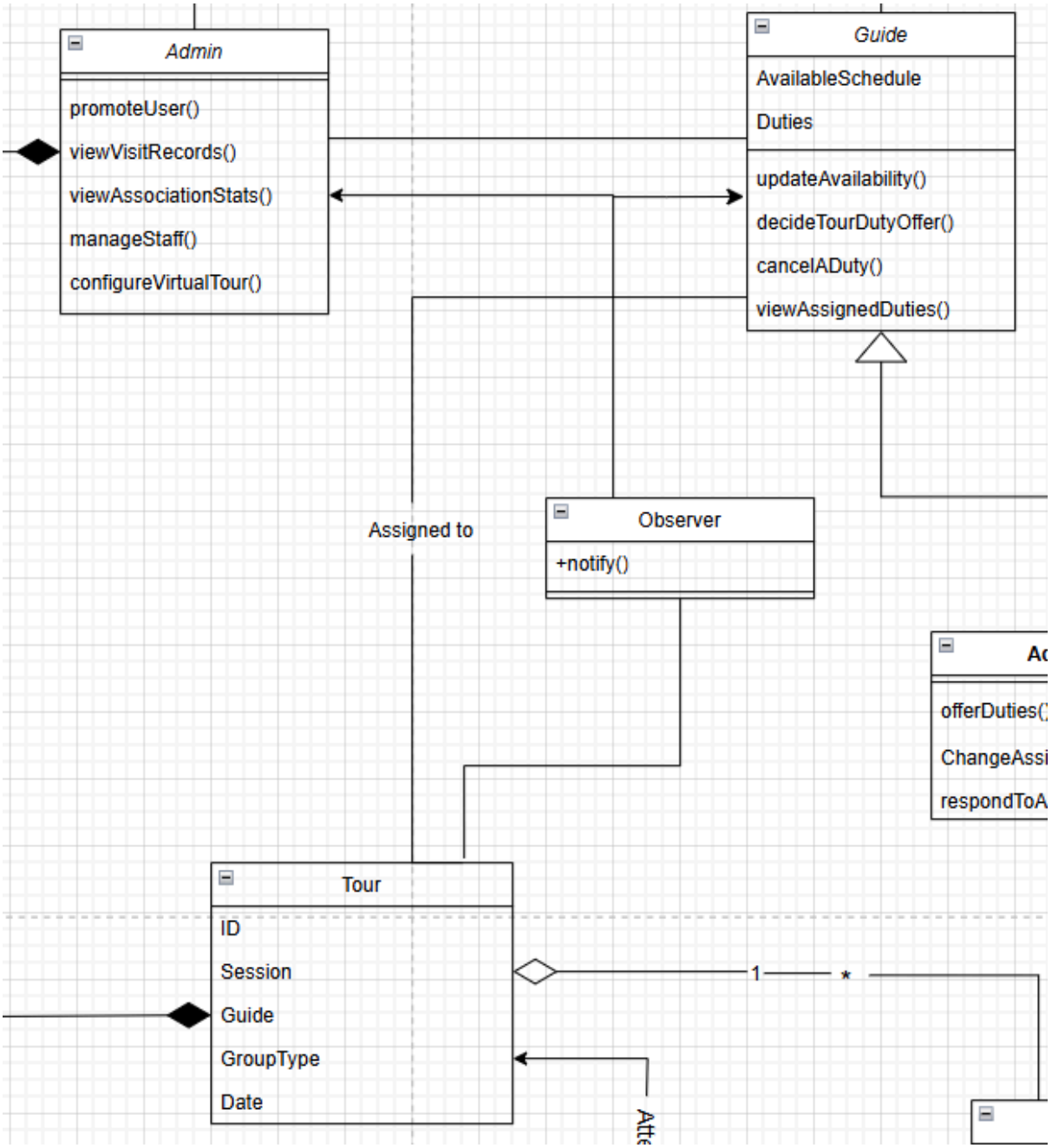
- Scenario: In your system, when a Tour receives feedback, you might want to notify the Guide, Admin, or other relevant parties.
- Problem: Without the observer pattern, the Tour class would need to manage the notification process, which violates the principle of low coupling.
- Solution: Using the Observer Pattern, the Tour (subject) doesn't need to know who will receive the notifications. The Users, Admin, and Guide classes act as observers, and they get notified when the state of the Tour changes (e.g., when new Feedback is added).

Components:

1. Subject: `Tour`
    - Knows and maintains a list of its observers.
    - Notifies observers when the state of the `Tour` changes (e.g., when feedback is added).
2. Observers: `Guide`, `Admin`, `User`
    - These are interested parties that need to be updated when a Tour's state changes (for instance, when feedback is added).

Implementation:

- Subject: `Tour` will implement the method notify observers.


Admin
promoteUser()
viewVisitRecords()
viewAssociationStats()
manageStaff()
configureVirtualTour()
Guide
AvailableSchedule
Duties
updateAvailability()
decideTourDutyOffer()
cancelADuty()
viewAssignedDuties()
Assigned to
Observer
+notify()
offerDuties()
ChangeAssi
respondToA
Tour
ID
Session
Guide
GroupType
Date
1
*

2. Mediator Pattern

The Mediator Pattern is used to reduce the complexity of communication between multiple objects or classes by introducing a mediator object that centralizes the communication. In the context of your system, a TourManager could act as a Mediator to facilitate communication between objects like Guide, Tour, and Feedback, without them needing to know about each other directly.

Explanation:

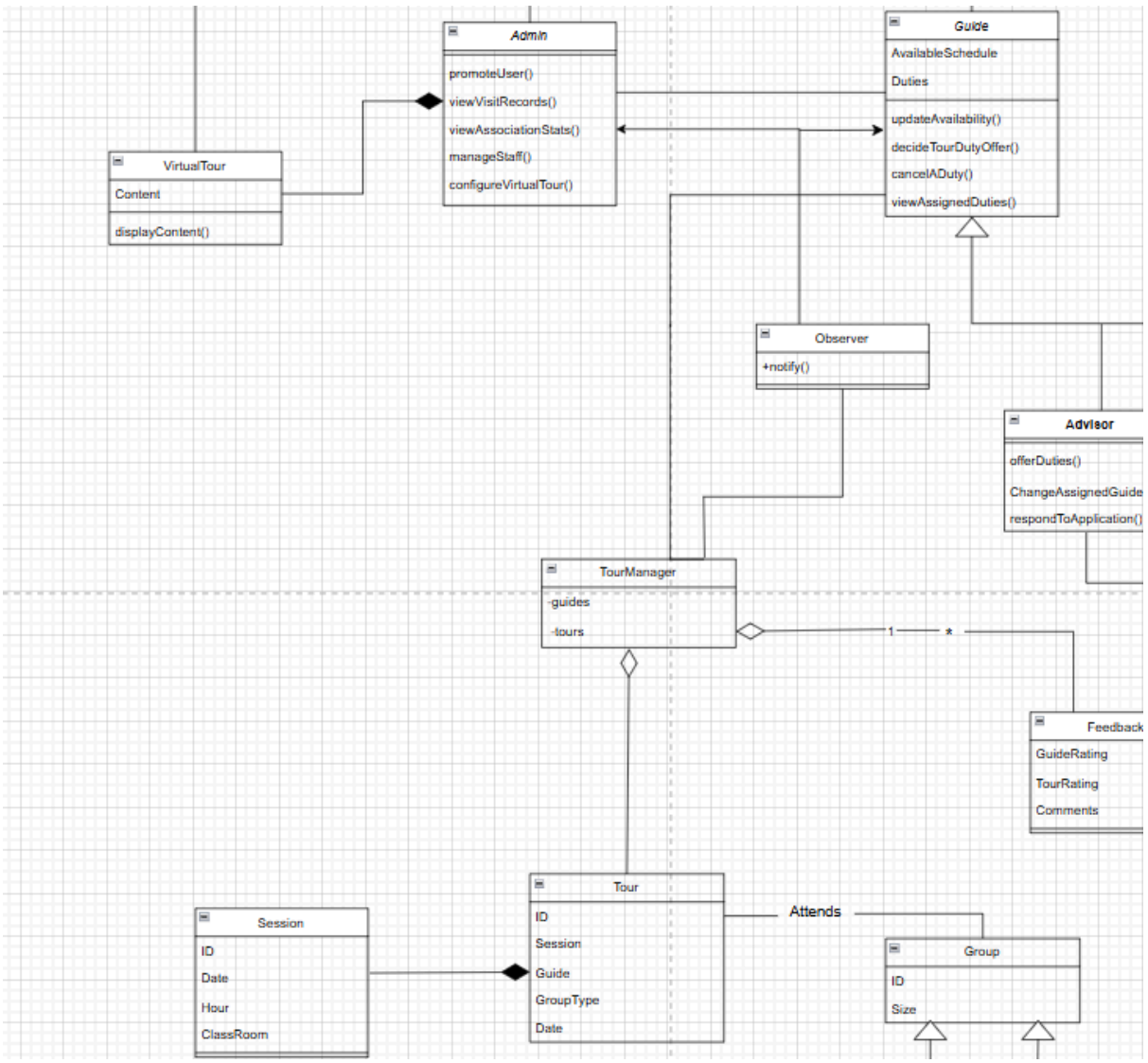
- Scenario: In your system, various objects like Guide And Tour need to interact with each other (e.g., assigning a guide to a tour, submitting feedback for a tour). Without the Mediator Pattern, each object would need to reference others directly, which increases the complexity and interdependency.
- Problem: With direct communication, objects become tightly coupled, leading to harder maintenance, testing, and scalability.
- Solution: Using the Mediator Pattern, a central Mediator object (like TourManager) can handle the interactions between different components, ensuring that objects do not need to directly reference each other. They only communicate with the Mediator, reducing coupling.

Components:

1. Mediator: `TourManager`
    - The class responsible for handling interactions between different components (e.g., assigning guides to tours, managing feedback).
2. Colleagues: `Guide`, `Tour`, `Feedback`, `User`
    - These classes interact through the Mediator rather than directly with each other.

Implementation:

- The TourManager class (Mediator) will manage the communication between components. For example, when a Guide is assigned to a Tour, the TourManager will handle the assignment rather than the Guide and Tour directly communicating.
- The Guide, Tour, and User classes will communicate only with the TourManager, which will ensure the proper coordination between them.


Admin
promoteUser()
viewVisitRecords()
viewAssociationStats()
manageStaff()
configureVirtualTour()
Guide
AvailableSchedule
Duties
updateAvailability()
decideTourDutyOffer()
cancelADuty()
viewAssignedDuties()
VirtualTour
Content
displayContent()
Observer
+notify()
Advisor
offerDuties()
ChangeAssignedGuide
respondToApplication()
TourManager
-guides
-tours
1
*
Feedback
GuideRating
TourRating
Comments
Tour
ID
Session
Guide
GroupType
Date
Attends
Session
ID
Date
Hour
ClassRoom
Group
ID
Size

Revised Class Diagram