CS319

S2T1- SOOF D5

Ömer Bıçakcıoğlu
Muhammed Furkan Başıbüyük
Sümeyye Acar
Ömer Edip Aras

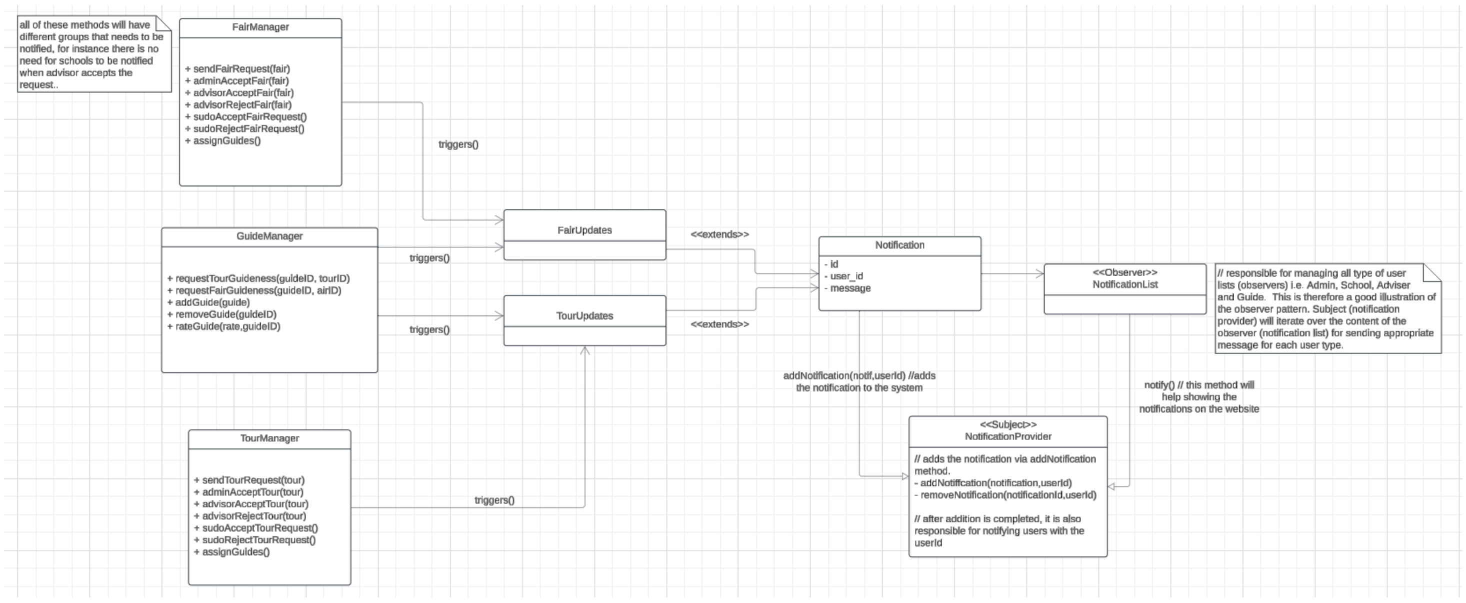
Design Patterns

We have used a variety of design patterns in the project implementation so far, but we will only show 2 of these design patterns: Observer and Facade Patterns

Observer Pattern

The observer pattern is a behavioral design pattern where an object (subject) maintains a list of observers (other objects). When the subject's state changes, it notifies all of the observes accordingly. We are using the Observing Pattern in our Notification System that updates Guides, Admins Advisors and schools for certain events. This use is intuitive due to the fact that a single event (subject) should update lists of several different user roles (objects).For instance when Advisor assigned a tour for a guide, guide that has been assigned has been notified by email. Detailed information can be found in the notes parts of the structure below.

Here's the structure:


all of these methods will have different groups that needs to be notified, for instance there is no need for schools to be notified when advisor accepts the request..
FairManager
+ sendFairRequest(fair)
+ adminAcceptFair(fair)
+ advisorAcceptFair(fair)
+ advisorRejectFair(fair)
+ sudoAcceptFairRequest()
+ sudoRejectFairRequest()
+ assignGuides()
triggers()
GuideManager
+ requestTourGuideness(guideID, tourID)
+ requestFairGuideness(guideID, airID)
+ addGuide(guide)
+ removeGuide(guideID)
+ rateGuide(rate,guideID)
triggers()
triggers()
FairUpdates
TourUpdates
<<extends>>
<<extends>>
TourManager
+ sendTourRequest(tour)
+ adminAcceptTour(tour)
+ advisorAcceptTour(tour)
+ advisorRejectTour(tour)
+ sudoAcceptTourRequest()
+ sudoRejectTourRequest()
+ assignGuides()
triggers()
Notification
- id
- user_id
- message
<<Observer>> NotificationList
// responsible for managing all type of user lists (observers) i.e. Admin, School, Adviser and Guide. This is therefore a good illustration of the observer pattern. Subject (notification provider) will iterate over the content of the observer (notification list) for sending appropriate message for each user type.
addNotification(notif,userId) //adds the notification to the system
notify() // this method will help showing the notifications on the website
<<Subject>> NotificationProvider
// adds the notification via addNotification method.
- addNotiffcation(notification,userId)
- removeNotification(notificationId,userId)
// after addition is completed, it is also responsible for notifying users with the userId

(please contact us if the diagrams are not legible)

NotificationProvider (Subject):

This is the main component that manages notifications in the system. It has methods like addNotification and removeNotification to handle the addition and removal of notifications.

When something changes, it notifies all the relevant users (observers) by triggering the notify() method. This is the core of the Observer Pattern.

NotificationList (Observer):

This represents the list of all observers who are interested in receiving notifications. Observers can include Admins, Schools, Advisors, and Guides.

Whenever the NotificationProvider triggers a notification, it iterates over the NotificationList to deliver updates to these observers.

FairManager, GuideManager, and TourManager:

These classes are responsible for handling specific types of actions in the system. For example, FairManager handles requests related to fairs, GuideManager manages guides, and TourManager deals with tours.

Each of them has methods like sendFairRequest, advisorAcceptFair, or requestTourGuideness, which may generate updates. These updates are sent to the NotificationProvider to notify relevant users.

FairUpdates and TourUpdates:

These classes extend the notification functionality for specific contexts, such as fairs or tours. They provide a way to specialize the notification process for different types of events.

Notification:

This represents the actual notification itself. It contains basic details like an id (to uniquely identify the notification), a user_id (to specify who should receive it), and a message (the content of the notification).

Flow of Notifications:

When an event happens (e.g., an advisor accepts a fair request), the relevant manager (e.g., FairManager) triggers the notification process.

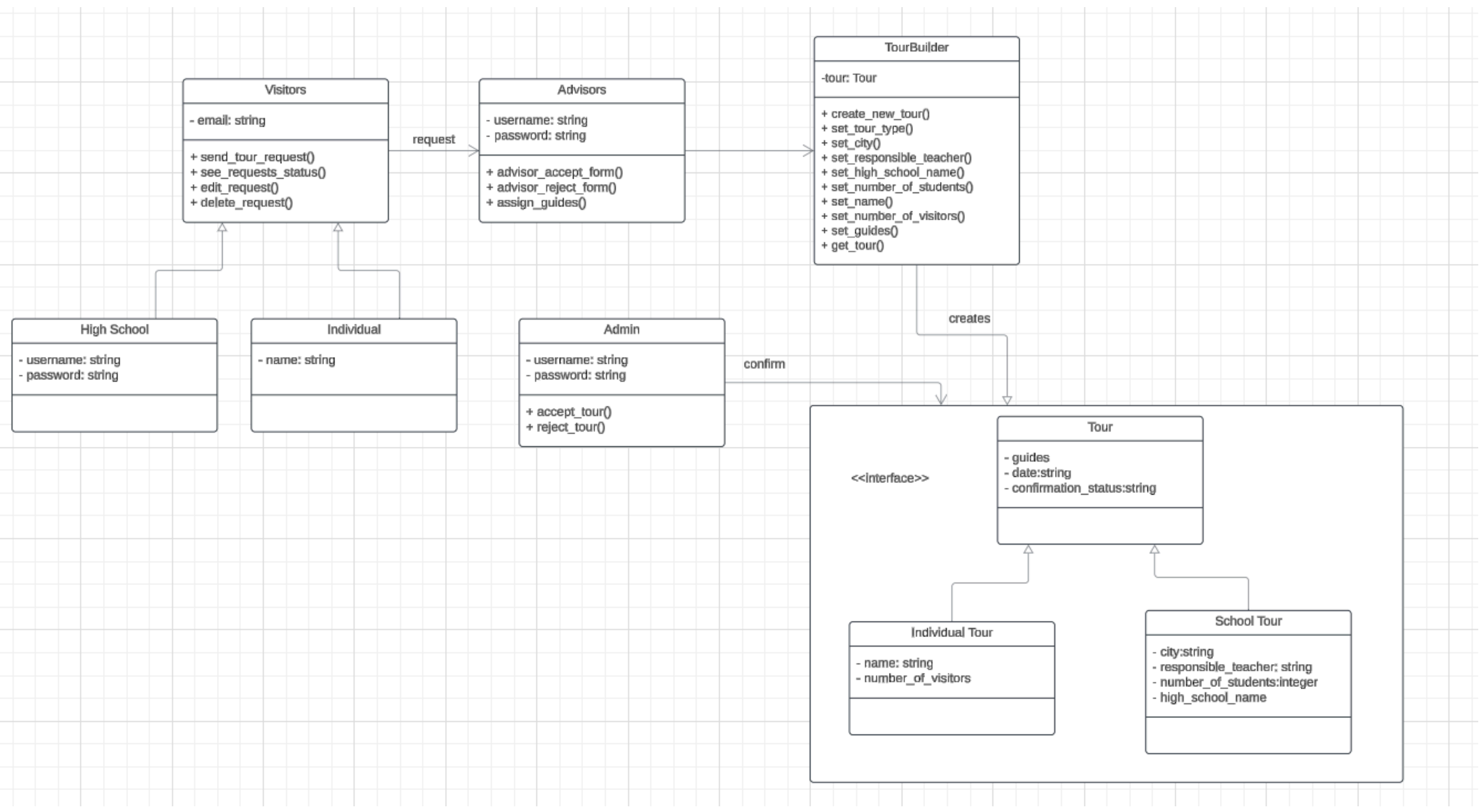
The NotificationProvider collects this information and uses the NotificationList to determine which observers need to be notified.

Observers like Admins, Schools, or Guides are then notified and can display or process the notification using the notify() method.

Builder Design Pattern

We use the builder design pattern for tours because it offers a way to construct complex objects step by step. Decoupling the construction process from the representation allows for different representations using the same construction process. In our case, the tour object represents the built product, and the pattern will be implemented there with clear roles and responsibilities.

Here's the structure:



Product (Tour):

Represents the final object being constructed.

Subclasses:

School Tour: Specifically designed for tours involving groups from high schools. Attributes include details such as the school's name, location, and the responsible teacher.

Individual Tour: Represents tours applied for by individual participants.

Builder (TourBuilder):

Defines the step-by-step process to construct a Tour object.

Responsible for creating the tour based on the form submitted and approved by the advisor, as well as the assignment of guides by the advisor.

Concrete Builder (Advisors):

Acts as an intermediary, evaluating the feasibility of the requested tour based on the information provided by the visitor.

If the tour is feasible, the advisor approves it and triggers the TourBuilder to create the corresponding Tour object.

Example: An advisor may construct a School Tour by inputting details like the high school's name, location, and the teacher responsible for the group.

Client (Visitors):

The entity that initiates the tour creation process.

Visitors do not interact directly with the TourBuilder. Instead, they submit a tour request, which is handled by advisors to trigger the building process.

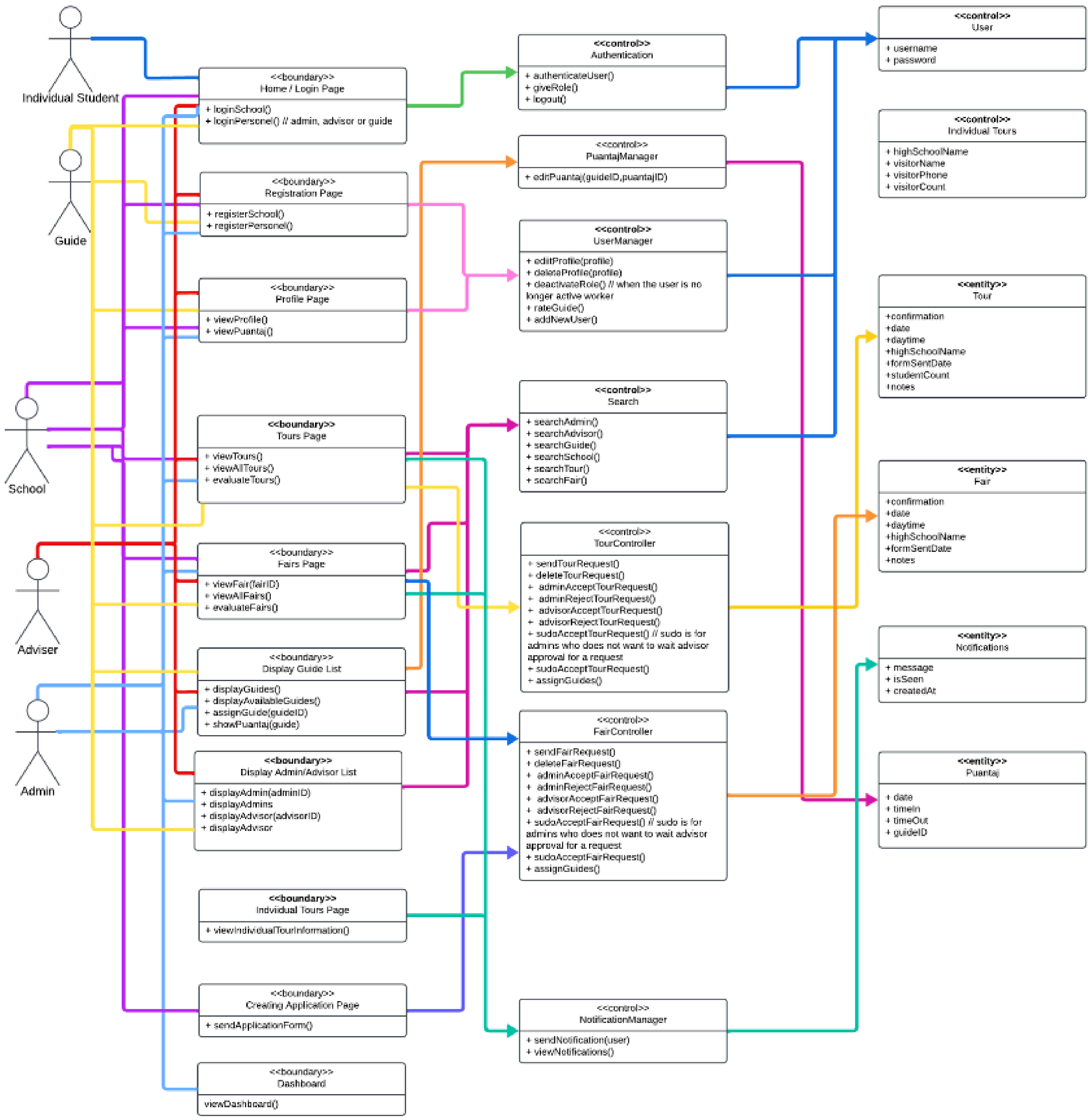
Admin:

The final approver or validator of the constructed Tour object.

Once the tour is built, the admin reviews it and either confirms the tour using accept_tour() or rejects it using reject_tour().

Updated Class Diagrams

since all requirements were met, we did not update the domain / class diagrams.



(if not visible, feel free to contact us.)