Design Document

Group 7, Project 13

Cab Sharing

Mahidhar Alam, Rithvik Kondapalkala,

BVS Gnaneswar, Bhanu Prakash S

5th April 2023

Contents

1 Introduction

The project's main objective is to build a portal for the **Cab Sharing** facility for the IIT Hyderabad residents. In This Document we will take you through the Modules, Class Diagrams, Implementations of the project, sequence diagrams for Join and Unjoin Use Cases and Design Patterns used in this Project.

2 Modules

The entire system is divided into code units as follows:

1. User
2. UserViews
3. MyBooking
4. Schedule
5. ActiveUser

2.1 Participants/ Functionalities table

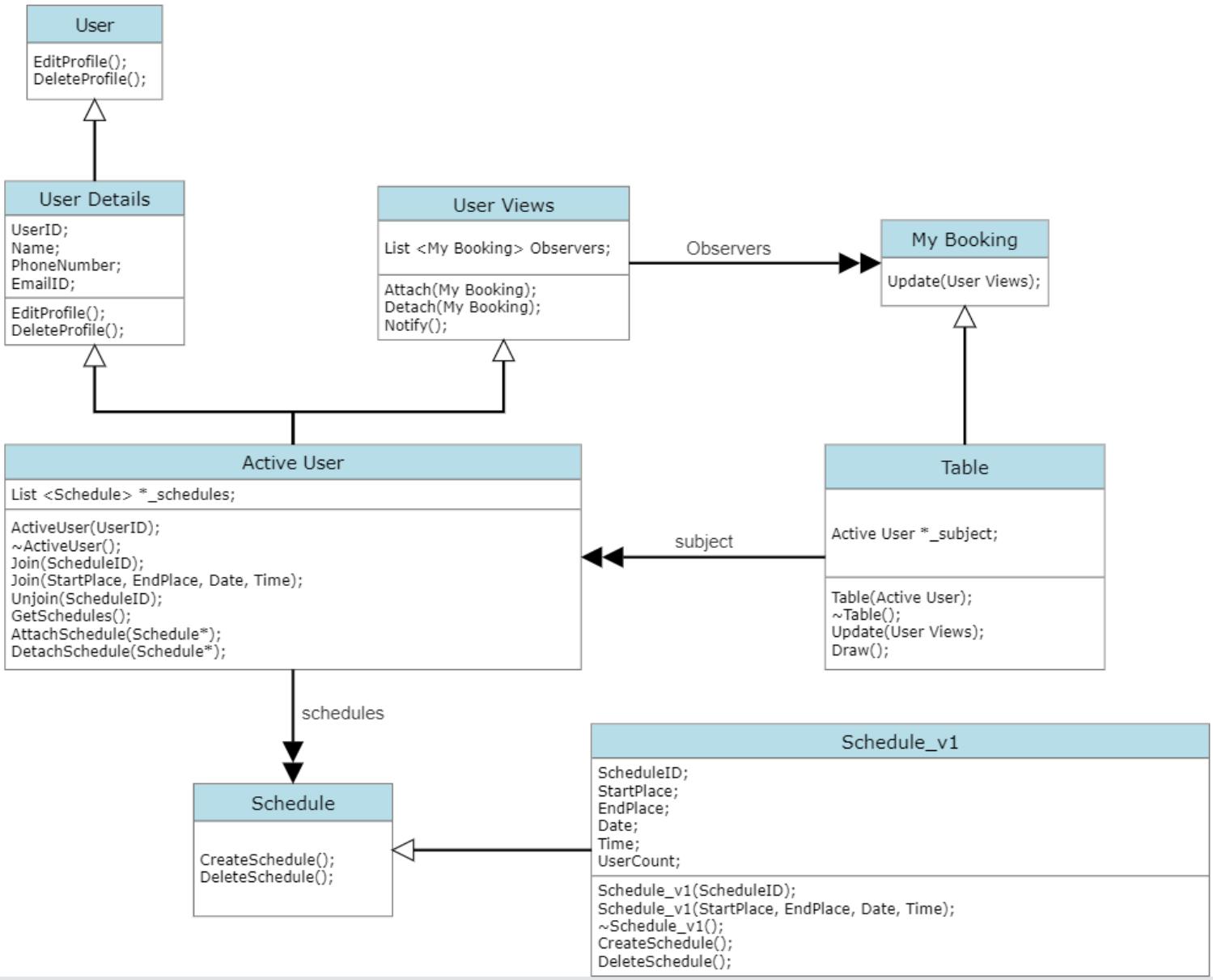
.

Class	Functionality
User	1. Provides an interface for editing and deleting the profile.
User Details	1. Stores the user details and provides the functionalities of edit and delete profiles of User class.
User Views	1. Knows its observers. Any number of My Booking objects may be present in User Views. 2. Provides an interface for attaching and detaching My Booking objects.
My Booking	1. Defines an updating interface for objects that should be notified of changes in the User Views.
Table	1. Maintains a reference to a Active User object. 2. Stores the state that should stay consistent with the User View. 3. Implements the observer updating interface to keep its state consistent with the User View.
Schedule	1. Provides an interface for creating and deleting the schedule.
Schedule_v1	1. Stores the details of the schedule. 2. Provides the functionalities of the creating and deleting the schedules of Schedule class.
ActiveUser	1. It maintains the list of schedules in which the user is present. 2. It provides functionalities for joining and unjoining the table. 3. Stores the state of interest to Table objects. 4. Sends a notification to each observer when the state changes.

Table 1: Functionality Table

3 Class Diagrams

3.1 Structure



Design Diagram

4 Implementation

In this section, you will get to know the reasons behind each abstraction and design we have used, keeping the mind future requirements.

1. User is an abstract class which have editProfile and deleteProfile Functions where UserDetails overides them .
2. MyBooking is an abstract class as an observer which has view of my booking on screen as tabular data etc
3. Schedule is an abstract class containing createSchedule and DeleteSchedule functions and which is overrided by the its children ScheduleV1 .

4.1 Relations

4.1.1 Abstraction

- User is an abstraction of UserDetails
- MyBooking is an abstraction for View of Schedules

4.1.2 InHertiance

- UserDetails Inherits User
- ActiveUser Inherits UserDetails and UserViews.
- Table Inherits MyBooking
- ScheduleV1 inherits Schedule

4.1.3 Association

- UserViews and MyBooking have an association where UserViews contains a List of MyBooking Observers
- ActiveUser and Schedule have an association relation where ActiveUser contains the list of schedules
- Table and ActiveUser have an association relation where Table contains the Active User of the list of schedules

4.2 Future Possible Extensions

- There can be a new view of showing My booking in future using graphs and other representations so we have used an observer pattern for it
- Schedule Class can have other details like Intermediate station or Time period which is flexible for them so ScheduleV1 is the present ConcreteClass for Schedule may be in the future we could change the implementation and work accordingly.

5 Sample Code

5.1 UserDetails

Below is the Implementation of User and UserDetails

```
class User
{
public:
    virtual void EditProfile() = 0;
    virtual void DeleteProfile() = 0;

}

class UserDetails : public User
{
public:
    int userId;
    string name;
    int phoneNumber;
    string email;
    void EditProfile(string name = name, int phoneNumber =
    phoneNumber, string email = email);
    void DeleteProfile();

}

void UserDetails::EditProfile(string name = name, int phoneNumber =
     phoneNumber, string email = email)
{
    // Using userId change the Details in the Database
}

void UserDetails::DeleteProfile();
{
    // Remove the UserId from User Database
    // Remove the UserId from Merge Database
    // Logout of the session so that all objects are deleted
}
```

5.2 Schedule

Below is the Implementation of Schedule and ScheduleV1

```
class Schedule
{
    public:
    virtual void CreateSchedule()=0;
    virtual void DeleteSchedule()=0;
}
class Schedule_V1:public Schedule
{
    public:
    int scheduleId;
    string startPlace;
    string EndPlace;
    String Date;
    string Time;
    int userCount;
    Schedule_V1(int scheduleId );
    Schedule_V1(string startPlace,string EndPlace,string Date,
    string Time);
    ~Schedule_V1();
    void CreateSchedule();
    void DeleteSchedule();

}
void Schedule_V1::CreateSchedule()
{
    //new scheduleId is created and added to Schedule Database
}
Schedule_V1::Schedule_V1(int scheduleId )
{
    //Data Retrived from Schedule Database
}
Schedule_V1::Schedule_V1(string startPlace,string EndPlace,string
    Date,string Time)
{
    //create object
    userCount=1;
    Schedule_V1::CreateSchedule();

}
void Schedule_V1::DeleteSchedule()
{
    //delete scheduleId from Schedule Database
    //delete schedule from Merge table
    ~Schedule_V1();
}
```

5.3 MyBooking

Below is the Implementation of MyBooking and Table

```
class MyBooking
{
    public:
    virtual void Update(UserViews*)=0;
}
class Table:public MyBooking
```

```
{
    public:
    Table(ActiveUser*);
    ~Table();
    void Update(UserViews*);
    void Draw();
    private:
    ActiveUser * _subject;

}
Table::Table(ActiveUser*s)
{
    _subject=s;
    _subject->Attach(this);
}
Table::~Table()
{

    _subject->Detach(this);
}
void Table::Update(UserViews* _theChangedSubject)
{
    if(_theChangedSubject==_subject)
    {
        Draw();
    }
}
void Table::Draw()
{
    List<Schedule> *Schedules=_subject->GetScchedules();
    //Draw;
}
```

5.4 UserViews

Below is the Implementation of UserViews

```
class UserViews
{
public:

   void Attach(MyBooking*);
   void Detach(MyBooking*);
   void Notify();
private:
    List<MyBooking*> *_observers;
}
void UserViews::Attach(MyBooking *o){_observers->Append(o);}
void UserViews::Detach(MyBooking *o){_observers->Remove(o);}
void UserViews::Notify()
{
    ListIterator<MyBooking*> i(_observers);
    for (i.First();!i.IsDone();i.Next())
    {
        i.CurrentItem()->Update(this);
    }
}
```

5.5 ActiveUser

Below is the Implementation of ActiveUser

```
class ActiveUser:public UserDetails,public UserViews
{
    private:
    List<Schedule> *_schedules;
    public:
    ActiveUser(int userId);

    ~ActiveUser();
    void Join(int scheduleId);
    void Join(string startPlace,string EndPlace,string Date,string
    Time);
    void AttachSchedule(Schedule *schedule);
    void DetachSchedule(Schedule *schedule);
    void UnJoin(int scheduleId);
    List<Schedule>* GetSchdeules();

}
ActiveUser::ActiveUser(int userId)
{
    //retrive Data from user  Database
    //retrive schedule from schedule Databse
    MyBooking * o=new Table(this);

}
void ActiveUser::AttachSchedule(Schedule *schedule)
{

    _schedules->append(schedule);
}
void ActiveUser::DetachSchedule(Schedule *schedule)
{
     _schedules->remove(schedule);
}
void ActiveUser::Join(int scheduleId)
{
    //Schedule of scheduleId from DB
    Schedule * schedule=new Schedule_V1(scheduleId);
    schedule.userCount++;
    AttachSchedule(schedule);
    // _schedules->append(schedule);
    AttachSchedule(scheduleId);

    //increment userCOunt in Database
    //Merge Database Update
    Notify();

}
void ActiveUser::Join(string startPlace,string EndPlace,string Date
    ,string Time)
{
    Schedule * schedule=new Schedule_V1(startPlace,EndPlace,Date,
    Time);
    schedule.userCount++;
    AttachSchedule(schedule);
```

```
52
53      //increment userCOunt in Database
54      //Merge Database Update
55      Notify();
56  }
57  void ActiveUser::UnJoin(int scheduleId)
58  {
59      //Schedule of scheduleId from DB
60      Schedule * schedule=new Schedule_V1(scheduleId);
61      schedule.userCount--;
62     DetachSchedule(schedule);
63
64      //decrement userCOunt in Database
65      //if schedule users count is 0 then Delete Schedule()
66      if(schedule.userCount==0)
67      {
68          schedule.DeleteSchedule();
69      }
70      else{
71
72          //Merge Database Update
73      }
74
75      Notify();
76
77  }
78  List<Schedule> *ActiveUser::GetSchdeules()
79  {
80      return _schedules;
81  }
```

6 Sequence or Activity Diagrams

We discuss 2 short but important use cases of the application.

6.1 Join Case

- This case deals with the scenario where a user wants to join an existing schedule group.

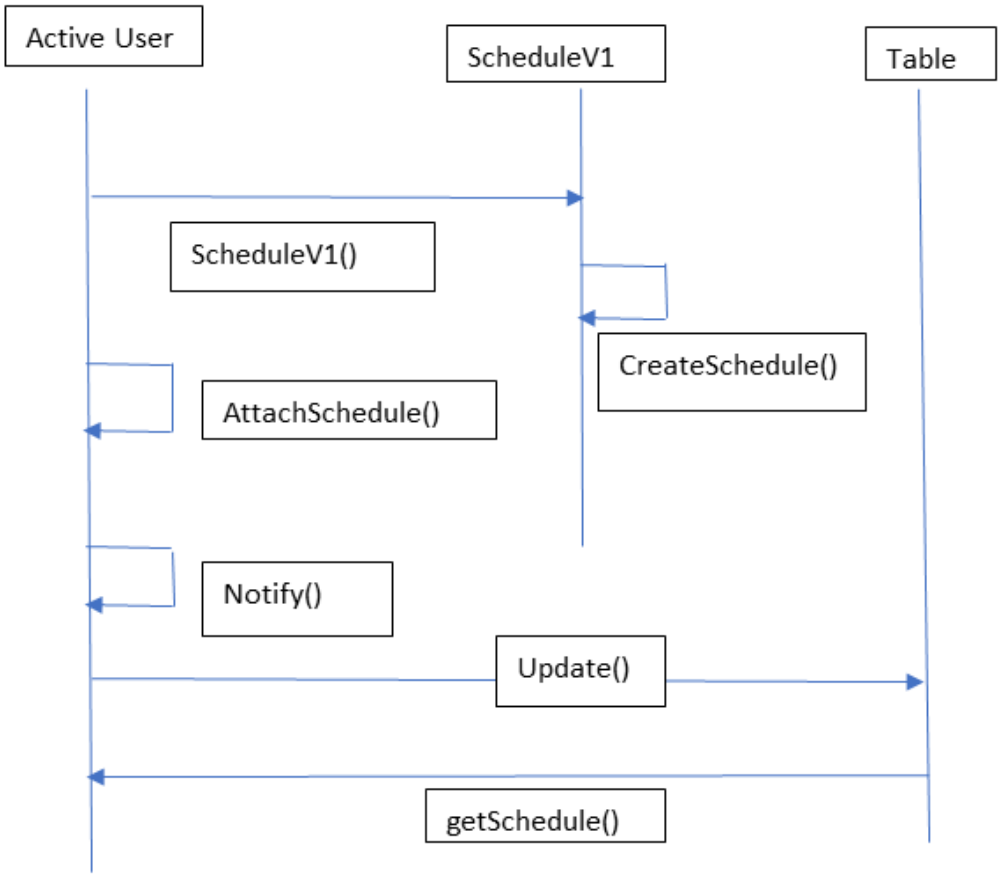


Figure 1: User joining a schedule group

6.2 UnJoin Case

- This case deals with the scenario where a user wants to unjoin or exit from a schedule group.

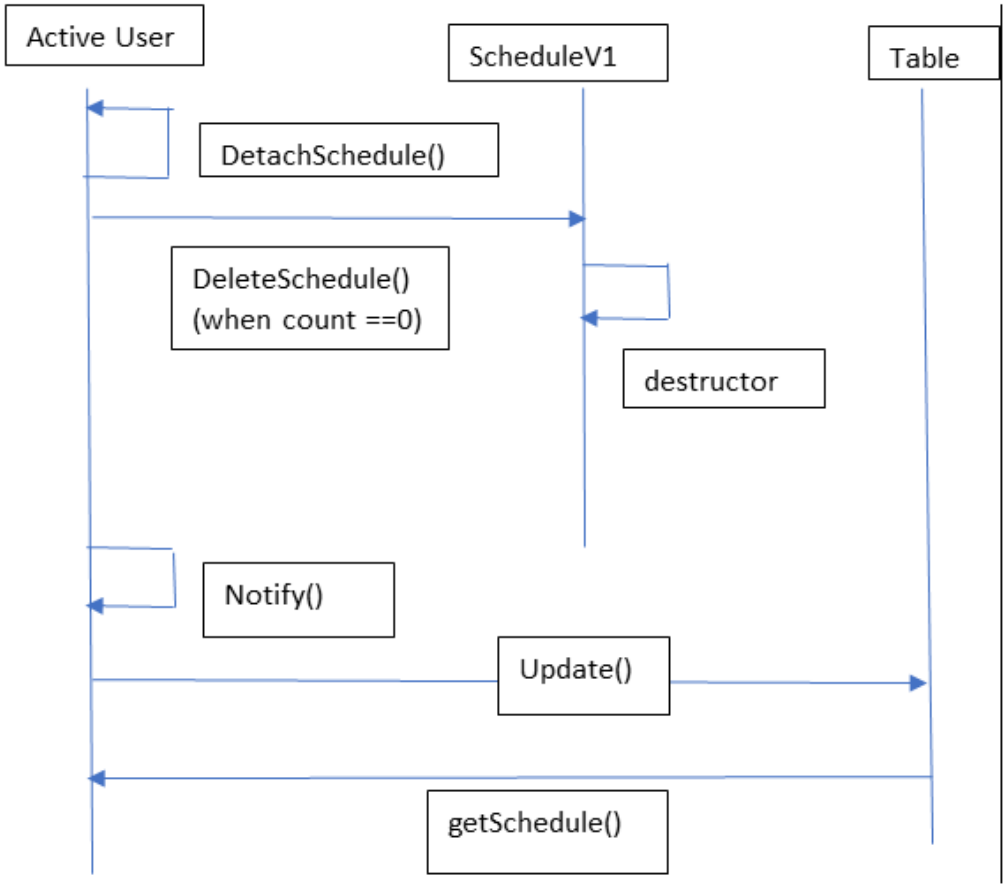


Figure 2: User unjoining or exiting a schedule group

7 Design Patterns Used

7.1 Observer Pattern

Here UserViews is the Subject, MyBooking is the Observer, ActiveUser is the Concrete Subject, and Table is Concrete Observer.

7.1.1 Structure

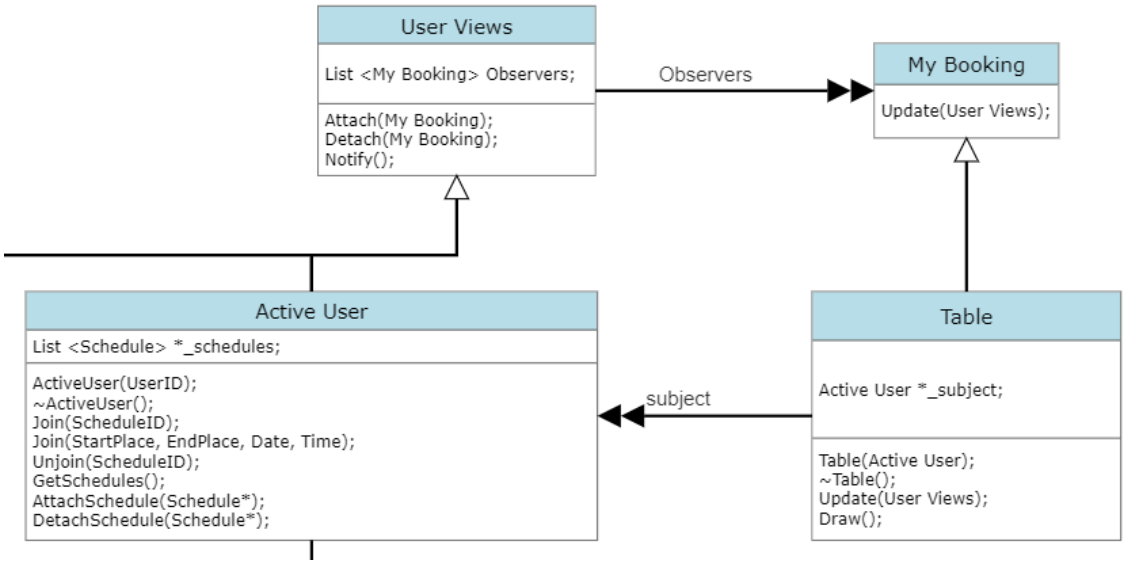


Figure 3: Observer Pattern

8 Functionalities Handled Using Existing APIS

8.1 Login /Sign Up

The **authentication flow** in the application, typically involves redirecting the user to the API's authentication endpoint, where they will be prompted to log in or grant access to the application. Once the user is authenticated, the API redirects them to the application with an authorization code or access token.

The **sign-up flow** typically involves the user entering their email address and choose a password and entering the User Details. You can then use the email address to check if the user already exists in your application's database and

also we have to check whether it belongs to IITH. If they do not exist, you can create a new user and store their information in the database. If it does not belong to IITH then we should not create the user

The **login flow** typically involves the user having to enter their email address and password. You can then use the email address to retrieve the user's information from the database and verify their password. If the credentials are correct, you can log the user in and redirect them to the Home page /View Page.

8.2 Role Management

Implementing Role or Permission field in API's User Management. While Logging, checking if a user is an admin or a normal user using APIs as the user roles or permissions are stored in the API's user management system.

8.3 Search Schedule

Designing a **Search bar** in the front end that allows users to enter their desired search criteria and submits the search query to the API. API responds to the search request. The front end will parse the request and show the desired result.

9 References

- Design Patterns, Gang Of Four