CSCI 5448 :Object - Oriented Analysis and Design

Team #34: FoodBuddy

Group Members:

Vikhyat Goyal
Pavan Dhareshwar
Rishi Soni

Vision

- Create a Restaurant and Food recommendation system which learns and betters itself by using feedbacks and user experience.

Objectives

- Project planning using structured methodologies such as identifying participants & use cases, creating use case charts, class diagrams, activity diagrams and sequence diagrams, adhering to the UML 2.0 standards.
- Identifying and implementing design patterns suitable for the project in Java
- Implementing the project functionality using the Spring MVC framework.

Project Description

- Created an application to generate restaurant and food suggestions based on user profile and past experiences.
- Recommendations generated on the basis of a points - based algorithm.
- Additionally, enabled the addition of new restaurants (hosts), modify their offerings, and promote their businesses.

Project Setup

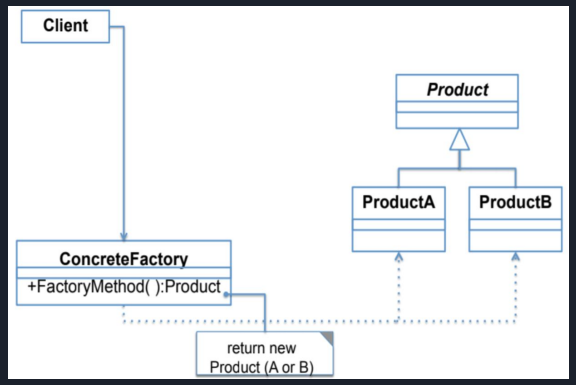
- Spring MVC framework and hibernate used to build the entire system.
- Controller class maps the requests to service layer methods.
- Database manager provides abstraction to find, insert, update and delete methods.
- MySQL database is used.

Design Patterns

- Three design patterns implemented:
  - Factory method
  - Observer
  - Iterator

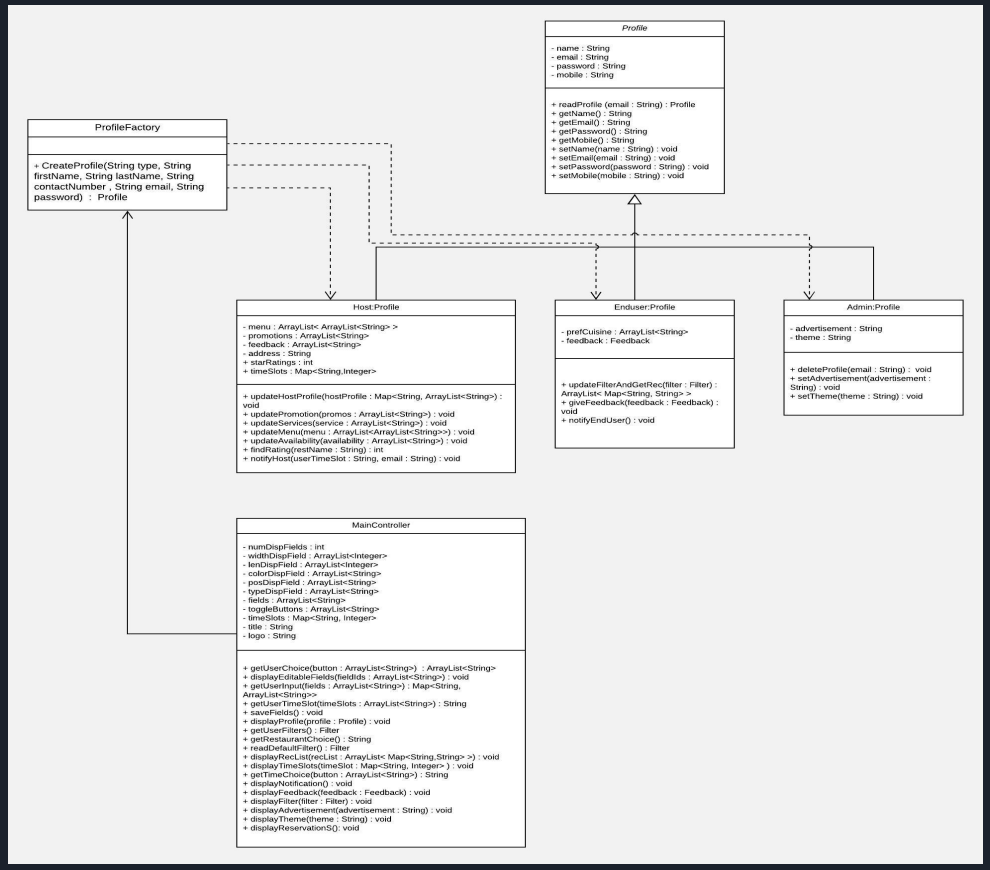
Factory Method

The system requires creation of new profiles which can be of any one of the types : Admin, Host, EndUser. This is the exact common problem which factory design pattern solves. We have added “ProfileFactory” class which depending on the kind of profile required creates a new object of either “Host” or “Admin” or “EndUser” type.



participant classes: ProfileFactory, Host, Admin, Enduser, MainController

Factory Method

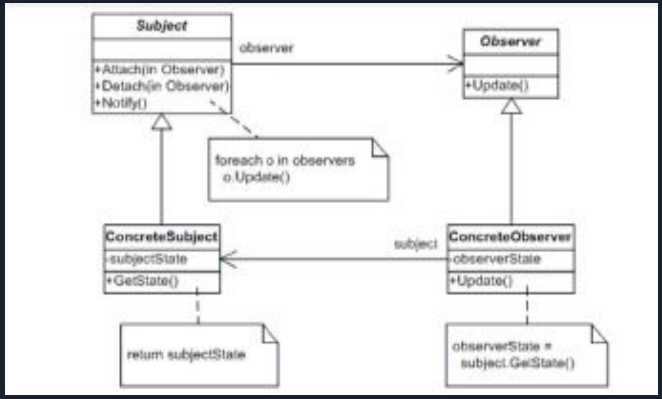



```
public void addProfile(ProfileFactory
                profileFactory, String type)  {

if (type.equals("endUser")) {
                        EndUser endUser =
profileFactory.getEndUser();
                        dbManager.saveToDb(endUs
er);
                }
else if (type.equals("host")) {
                        Host host = (Host)
profileFactory.getHost();
                        dbManager.saveToDb(host);
                }
else if (type.equals("admin")) {
                        Admin admin = (Admin)
profileFactory.getAdmin();
                        dbManager.saveToDb(admin
);
                }
}
```

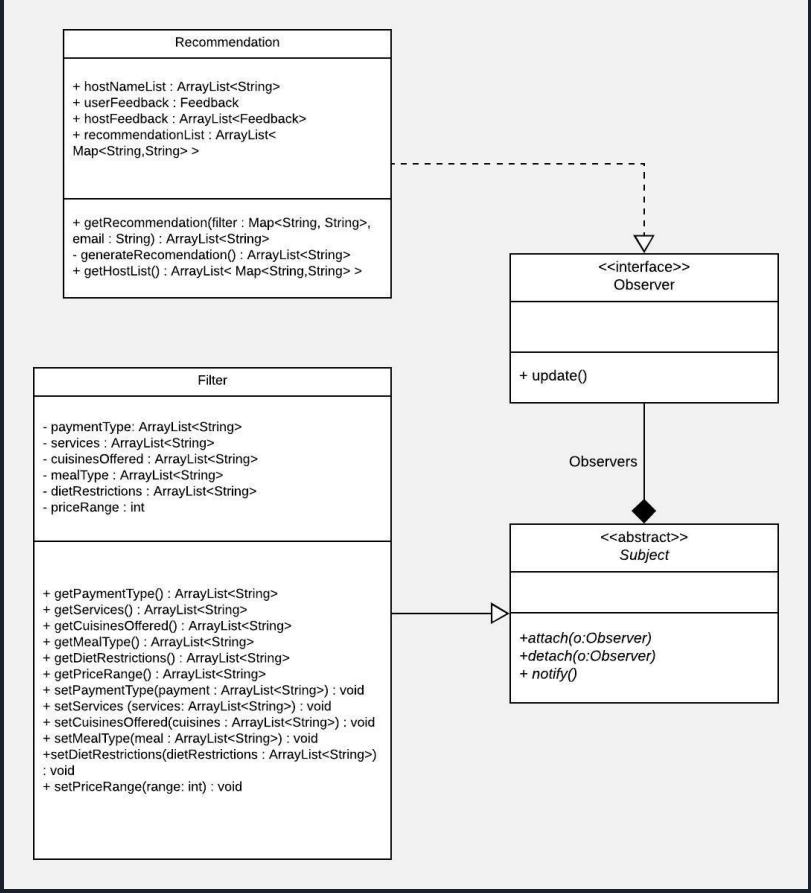
Observer

The system requires that whenever the Filter is updated, new Recommendations are generated. This problem is solved with the Observer design pattern. The Recommendation class acts as an observer, while the Filter class acts as a subject. The Filter class attaches itself to an observer object, which it uses to notify the Recommendation class that the filter has been updated. The Recommendation class itself waits on the Observer object for receiving the notification.



Participant classes: Recommendation, Observer, Filter, Subject

Observer




```
public class FilterManager extends
Observable{
public void setFilterById(EndUser
endUser, Filter filter){
      endUser.updateFilters(filter)
this.setEmail(endUser.getEmail());
            setChanged();
            notifyObservers();
}}
public class RecommendationManager
implements Observer{
String email =
((FilterManager)arg0).getEmail();
ArrayList<String> hostNameList =
getRecommendation(email);
Recommendation recommendation =
new Recommendation(hostNameList,
email);
dbManager.saveRecommendationsTo
Db(recommendation);
```

Demo Video