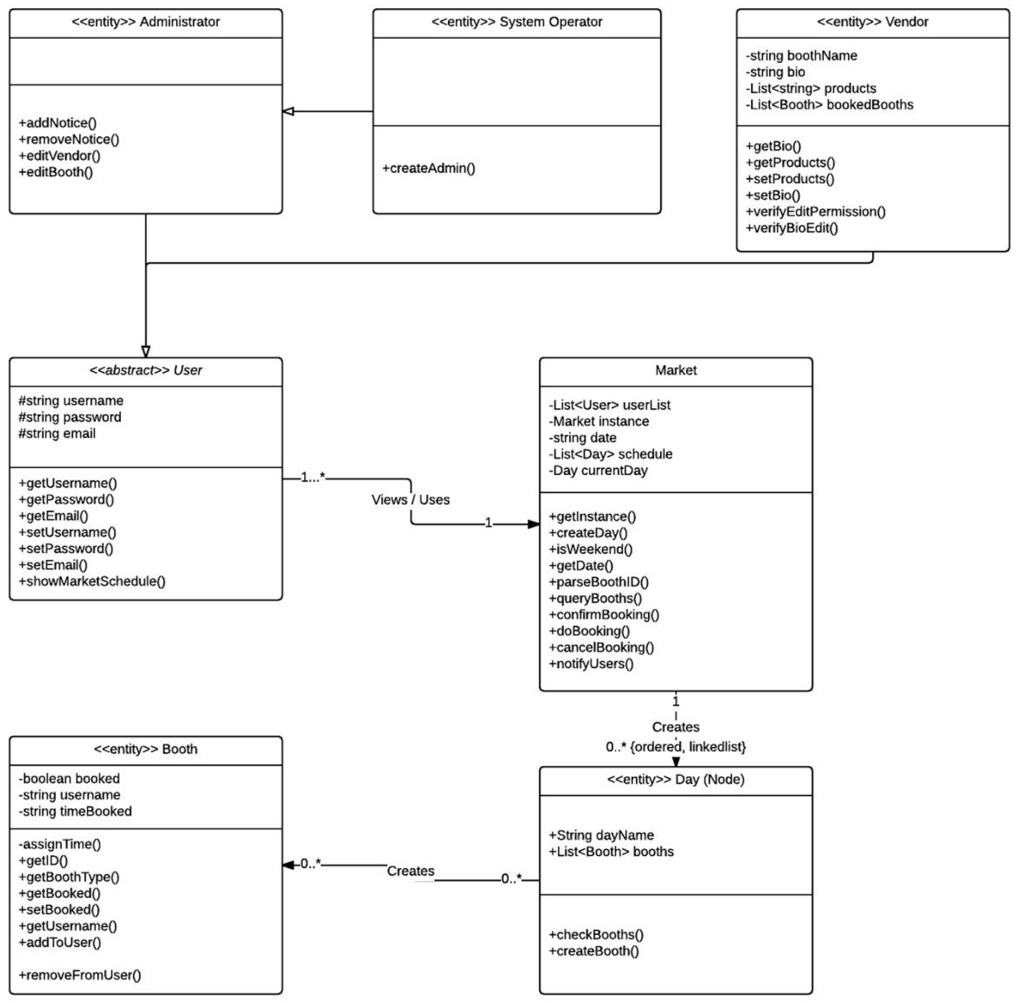
Class Diagram



1.0 Proposed Use of Design Patterns:

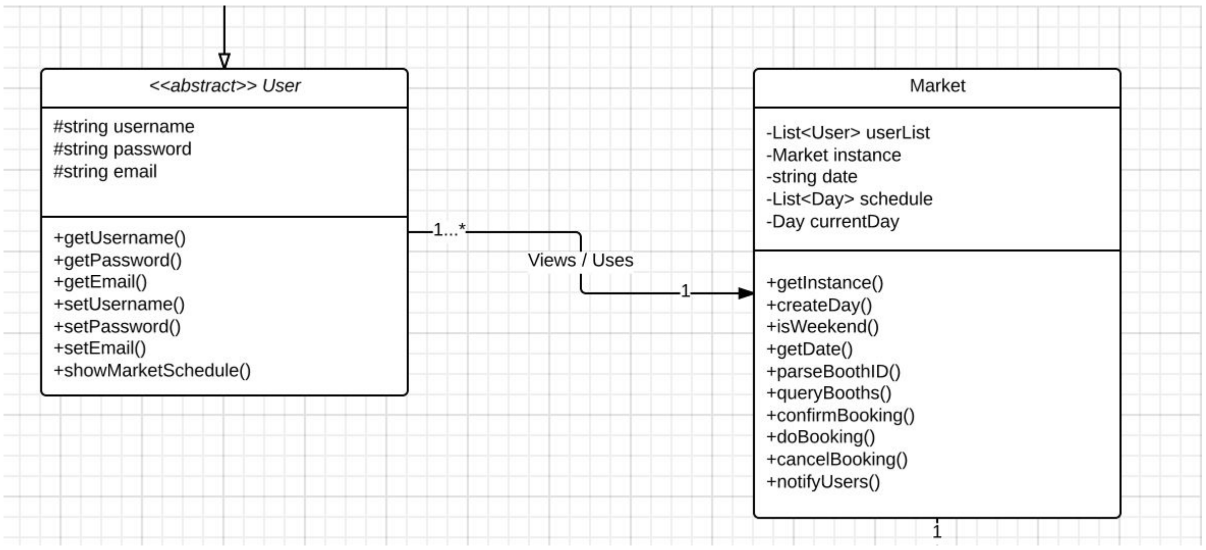
The Fernwood Farmers' Market Booth Scheduling System (FaBS) intends to use the Singleton, Facade and Observer patterns during implementation.

1.1 Singleton:

Our Singleton would be the `Market` class: the only class implementing the `getInstance()` method in the class diagram. The `Market` class consists of a list of `Day` objects, which in turn encapsulate all information relating to booth bookings (for that particular day) by storing an array of `Booth` objects. As we are only dealing with one market with regards to the problem definition, we currently only require one instance of the `Market` class, and hence the use of the Singleton design pattern.

1.2 Observer:

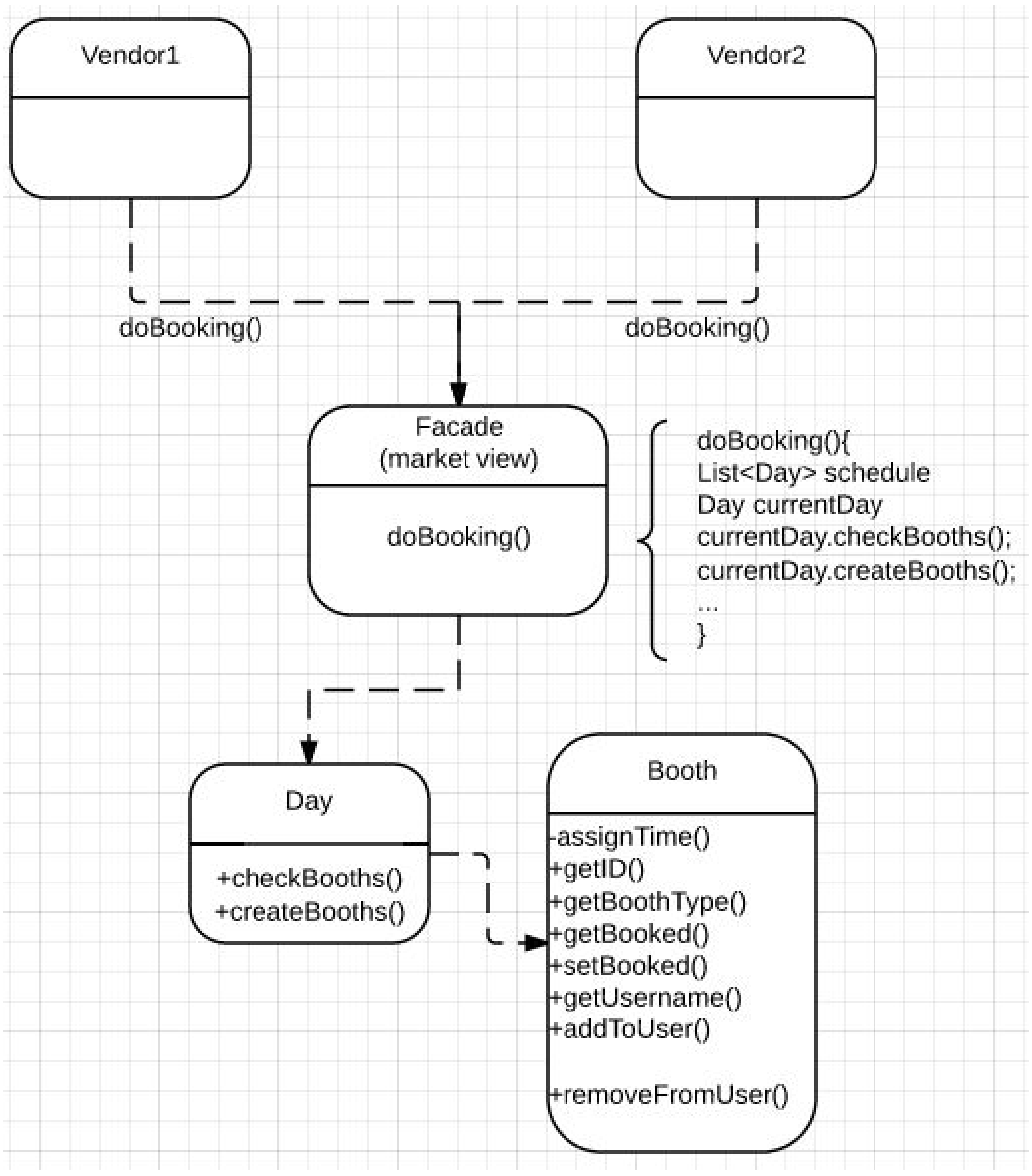
With regards to how our frontend UI works, all users are displayed with a view of the current day market schedule using a central market map which encompass all information relating to market booths and bookings. The map would contain colour-coded booths which allow the user to identify whether or not a time slot within that booth is bookable. These booth views would require real-time updates according to scheduling changes caused by real-time booking confirmations and cancellations. The Observer design pattern can be successfully applied here so that updates in the market schedule would be pushed to all users' pages in real-time.



With regards to our system, the `Market` class would hypothetically contain a list of its observers, i.e., `List<User> userList`, and would call the `notifyUsers()` function which would in turn call `User.showMarketSchedule()` to push any changes caused to the market schedule onto the user views.

1.3 Facade:

The user accesses all information through the market for all information calls (as shown in the class diagram), which classifies the system under the Facade design pattern. The Facade pattern hides the complexity of manipulating individual days and booth bookings in order to provide a simplified interface for the user. A typical vendor only needs to access their account information, see the schedule and make bookings. The underlying methods of parsing booths and creating schedules are hidden from the user. The figure below illustrates the booking abstraction between vendors and the market.


Vendor1
Vendor2
doBooking()
doBooking()
Facade
(market view)
doBooking()
doBooking(){
List<Day> schedule
Day currentDay
currentDay.checkBooths();
currentDay.createBooths();
...
}
Day
+checkBooths()
+createBooths()
Booth
-assignTime()
+getID()
+getBoothType()
+getBooked()
+setBooked()
+getUsername()
+addToUser()
+removeFromUser()