OOAD Report Part 6- Final Report
41- EMR System

Selva Priya Selvarathinavel, Selva Priyanka Selvarathinavel, Poorwa Hirve

1. List the features that were implemented (table with ID and title).

- **Business Requirements:**

ID	Title
BR001	Login must be through corporate email

- **User Requirements:**

ID	Title
UR001	Add new users to the system
UR003	Record patient problem list, patient history, vitals
UR004	Update medicine inventory
UR005	Diagnose a patient
UR011	Login to system
UR012	Maintain statistics
UR013	Update payments

- **Non Functional Requirements:**

ID	Title
NFR001	Medical records must be secure and access control must be maintained
NFR002	Patient records must be available to doctors without observable latency (< 10 ms)
NFR004	Any data from the database must be returned correctly or not at all. Wrong data is intolerable

2. List the features were not implemented from Part 2 (table with ID and title).

- **Business Requirements:**

ID	Title
BR002	Insurance officials currently involved with patients only have access to only their patients’ records
BR003	Employee payments are distributed last working day of each month
BR004	All employees can avail leave as per their prescribed leaves with a notice of one week

- **User Requirements:**

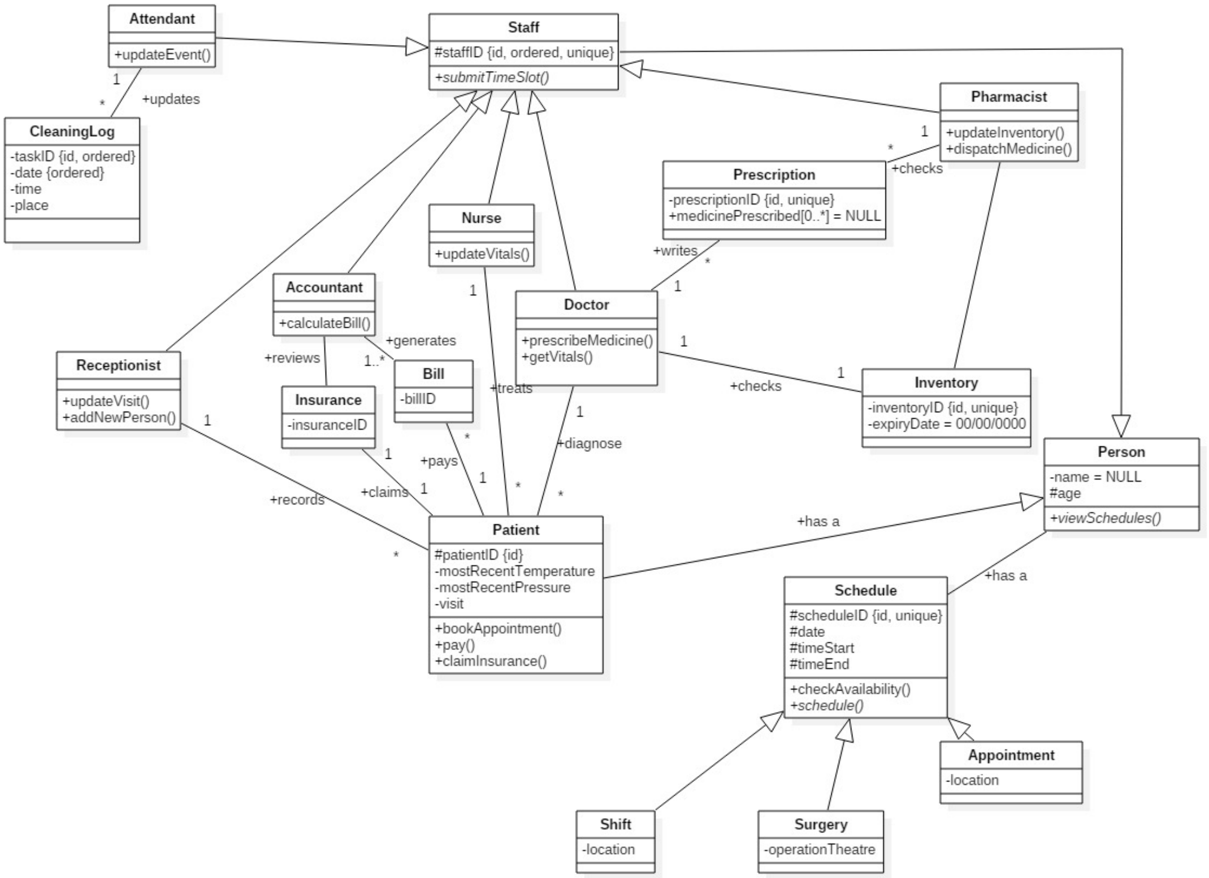
ID	Title
UR002	Claim Insurance
UR006	Schedule doctor appointment, surgeries, shifts
UR007	Access shift times and schedules
UR008	Keep track of periodic cleaning
UR009	Pay bills
UR010	Track payments

- **Non Functional Requirements:**

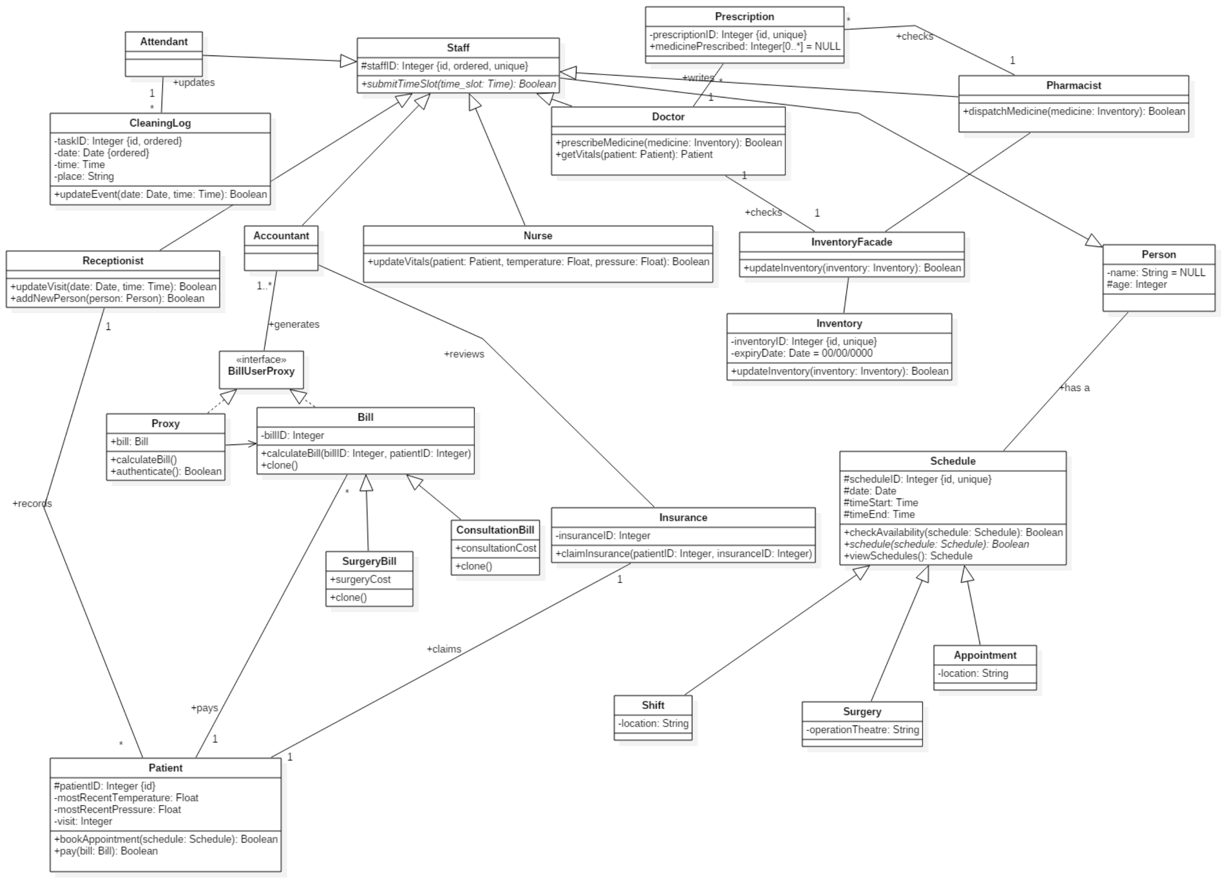
ID	Title
NFR003	Patients must be able to schedule an appointment using a calendar at the first page on the dashboard

3. Show your Part 2 class diagram and your final class diagram.
What changed? Why? If it did not change much, then discuss how doing the design up front helped in the development.

Old class diagram in part 2:


Attendant
+updateEvent()
1
*
+updates
CleaningLog
-taskID {id, ordered}
-date {ordered}
-time
-place
Staff
#staffID {id, ordered, unique}
+submitTimeSlot()
Pharmacist
+updateInventory()
+dispatchMedicine()
1
*
+checks
Prescription
-prescriptionID {id, unique}
+medicinePrescribed[0..*] = NULL
+writes
*
1
Nurse
+updateVitals()
1
Accountant
+calculateBill()
+generates
1..*
+reviews
Doctor
+prescribeMedicine()
+getVitals()
1
+checks
1
Inventory
-inventoryID {id, unique}
-expiryDate = 00/00/0000
Receptionist
+updateVisit()
+addNewPerson()
1
Bill
-billID
+treats
Insurance
-insuranceID
*
+pays
1
1
+diagnose
*
*
1
+claims
1
+records
*
Patient
#patientID {id}
-mostRecentTemperature
-mostRecentPressure
-visit
+bookAppointment()
+pay()
+claimInsurance()
+has a
Person
-name = NULL
#age
+viewSchedules()
+has a
Schedule
#scheduleID {id, unique}
#date
#timeStart
#timeEnd
+checkAvailability()
+schedule()
Appointment
-location
Shift
-location
Surgery
-operationTheatre

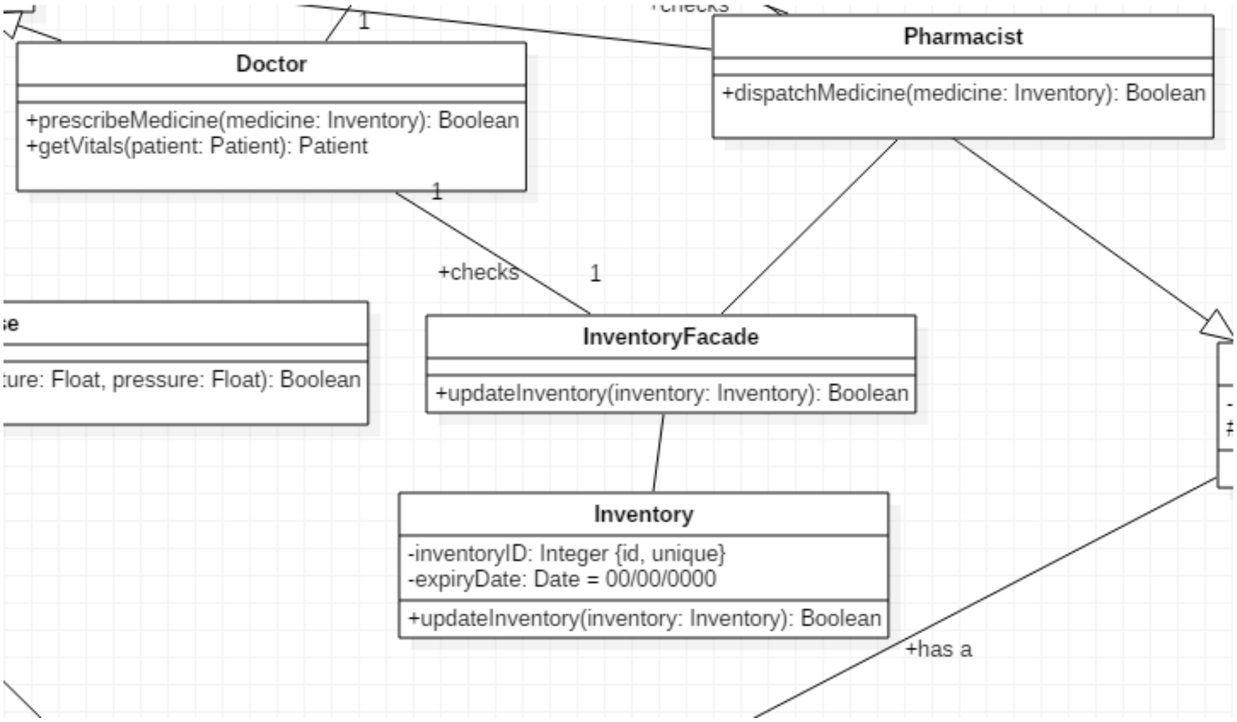
Final Class Diagram:


Attendant
Staff
#staffID: Integer {id, ordered, unique}
+submitTimeSlot(time_slot: Time): Boolean
Prescription
-prescriptionID: Integer {id, unique}
+medicinePrescribed: Integer[0..*] = NULL
*
+checks
1
Pharmacist
+dispatchMedicine(medicine: Inventory): Boolean
+updates
1
*
CleaningLog
-taskID: Integer {id, ordered}
-date: Date {ordered}
-time: Time
-place: String
+updateEvent(date: Date, time: Time): Boolean
+writes
*
1
Doctor
+prescribeMedicine(medicine: Inventory): Boolean
+getVitals(patient: Patient): Patient
1
+checks
1
InventoryFacade
+updateInventory(inventory: Inventory): Boolean
Person
-name: String = NULL
#age: Integer
Receptionist
+updateVisit(date: Date, time: Time): Boolean
+addNewPerson(person: Person): Boolean
Accountant
Nurse
+updateVitals(patient: Patient, temperature: Float, pressure: Float): Boolean
1..*
1
+generates
Inventory
-inventoryID: Integer {id, unique}
-expiryDate: Date = 00/00/0000
+updateInventory(inventory: Inventory): Boolean
+reviews
«interface»
BillUserProxy
+has a
Proxy
+bill: Bill
+calculateBill()
+authenticate(): Boolean
Bill
-billID: Integer
+calculateBill(billID: Integer, patientID: Integer)
+clone()
Schedule
#scheduleID: Integer {id, unique}
#date: Date
#timeStart: Time
#timeEnd: Time
+checkAvailability(schedule: Schedule): Boolean
+schedule(schedule: Schedule): Boolean
+viewSchedules(): Schedule
+records
*
ConsultationBill
+consultationCost
+clone()
Insurance
-insuranceID: Integer
+claimInsurance(patientID: Integer, insuranceID: Integer)
SurgeryBill
+surgeryCost
+clone()
1
+claims
Appointment
-location: String
Shift
-location: String
Surgery
-operationTheatre: String
+pays
1
*
1
Patient
#patientID: Integer {id}
-mostRecentTemperature: Float
-mostRecentPressure: Float
-visit: Integer
+bookAppointment(schedule: Schedule): Boolean
+pay(bill: Bill): Boolean

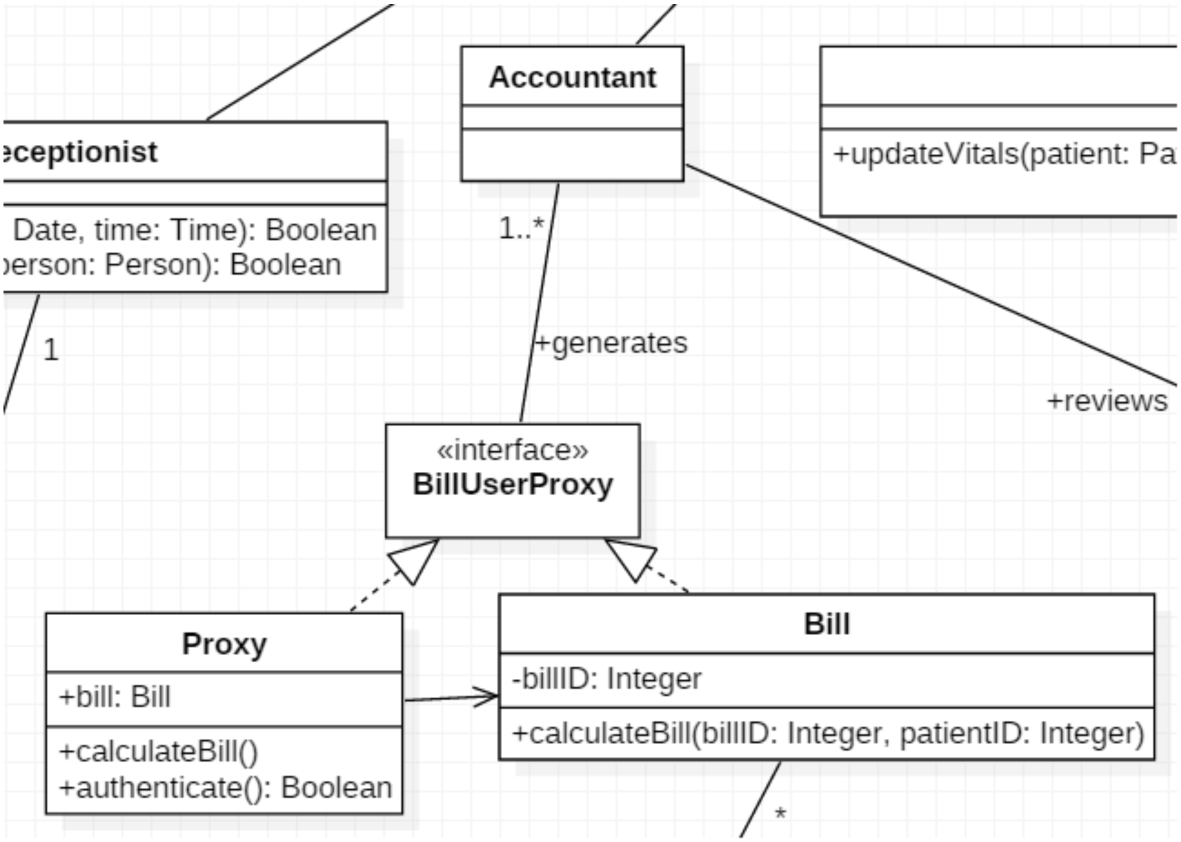
- Seen above are the old and the final class diagram.
- The old diagram did not mention all the correct attributes, methods and return types of the classes. This was corrected in the final class diagram.
- The associations and inheritance remained the same in the final class diagram compared to the old one.
- All attributes of every class were implemented as Models in Spring MVC.
- Methods performed by every actor were enclosed in Controllers for every class.
- As we were not familiar with Hibernate during the design of the old class diagram, few methods were placed incorrectly. For example, the updateInventory() method was placed in Pharmacist class, which was then considered to be a reasonable assumption. During the time of implementation using Spring MVC and Hibernate, Inventory class had its own getters and setters which could be used within the Inventory class. Therefore, it made more sense to keep the updateInventory() method in the Inventory class. The same applies to calculateBill() and viewSchedules(). These methods can be easily implemented by the getters and setters provided by Hibernate.
- The final change between the two class diagrams would be that the final class diagram implements few design patterns that enables us to incorporate their corresponding functionalities and benefits into our system.

4. Did you make use of any design patterns in the implementation of your final prototype?
If so, how? Show the classes from your class diagram that implement each design pattern (each design pattern as a separate image in the .PDF).
If not, where could you make use of design patterns in your system? Show a class diagram of how you could implement each design pattern and compare how it would change from your current class diagram.

- We made use of design patterns in our implementation.
- The first design pattern we implemented was the facade. This occurs around the Inventory class. This class is accessed by the Pharmacist and the Doctor. As the structure of the inventory is subject to change, in the absence of a facade, when the change occurs, the methods in Pharmacist and Doctor have to be rewritten to match the changing Inventory. To prevent this, a facade was implemented and ensured that the Inventory class is accessed seamlessly by the actors even if there are structural changes in its implementation. This design pattern follows the open close principle. It also supports the Interface Segregation Principle (SOLID) where even if the implementation structure of Inventory class changes, it wouldn't affect the usage of it by the Doctor and Pharmacist classes. Following is the part of the class diagram where facade was implemented.


+checks
1
Doctor
+prescribeMedicine(medicine: Inventory): Boolean
+getVitals(patient: Patient): Patient
1
Pharmacist
+dispatchMedicine(medicine: Inventory): Boolean
+checks
1
se
ture: Float, pressure: Float): Boolean
InventoryFacade
+updateInventory(inventory: Inventory): Boolean
Inventory
-inventoryID: Integer {id, unique}
-expiryDate: Date = 00/00/0000
+updateInventory(inventory: Inventory): Boolean
+has a

- The next design pattern we used was proxy. To authenticate the accountants' access to the bill. Only some accountants can access the bill of the patient. To protect this access, we used a “Protection Proxy.” Absence of the Proxy would allow all clerks who update bills to view the details of the accounts of the patient / hospital, which would be a violation of the patient’s rights and violation of hospital policies. The presence of Proxy ensures that the only the authorized accountants can access the invoices. This was implemented in Spring MVC to restrict access to certain users to access the view that helps in generating and viewing bills. Following is the part of the class diagram that shows the proxy design pattern.



5. What have you learned about the process of analysis and design now that you have stepped through the process to create, design and implement a system?

- We learned about the entire process that takes place in developing a production level software.
- We learnt various object oriented principles, object oriented designs and design patterns in class. A concept is truly learnt when it is applied. Therefore, we feel we have truly understood the concepts because they were extensively incorporated in our design.
- Most importantly, we understood the importance of the design phase of the project. Because, as Computer Science majors, we are taught only to code, it causes immense issues after implementation due to poor or no design. We realized this massive difference when we implemented our project with ease due to the amount of time we had spent on designing the system.
- Due to extensive designing of the project, the transition from design phase to development was seamless and easy.

- For all of us, Java was a new language to begin with. Though we had a hard time figuring out Spring MVC and Hibernate at the initial stages of development, we got a hang of it and were able to appreciate the capabilities of Java, Spring MVC and Hibernate.
- We also learnt to identify code smells and anti patterns and avoided them in our project for the intention of developing high quality code.