CSC 540 - Database Management Systems

Project Report 3

WolfHospital Management System

Team Members:
Mohd Sharique Khan (mkhan8@ncsu.edu)
Richa Dua (rdua2@ncsu.edu)
Siddu Madhure Jayanna (smadhur@ncsu.edu)
Viviniya Alexis Lawrence (palexis@ncsu.edu)

Assumptions

1. Registration Staff would be the admin of the system.
2. A nurse is responsible for all the beds in a ward.
3. While accepting the payment for the patient’s treatment, only one transaction method is used i.e. the payment can’t be split in multiple transactions. If the insurance company pays for the treatment, they pay in full.
4. Registration Fee is dependent on factors like arrival time of the patient into the hospital, emergency treatment.
5. No separate registration fee is charged to the patient. It is included in the consultation fee.
6. The system also allows patients to view their billing and medical records and update personal details like Name, Address etc.
7. For every new visit of a patient, a new billing record and medical record will be created.
8. Every billing record is associated with one medical record.
9. A patient can visit the hospital multiple times in a single day.
10. When a patient visits the hospital, a medical record and billing record for him will be created by registration staff. The registration staff will also assign a doctor to the patient.

Transactions

Use Case 1:
We are creating new Ward in the WolfHospital Management System using this method. Within the method, we also create the beds in each ward with respect to the capacity of each ward as entered by the registration staff.

Why implemented using Transaction:
If the ward creation fails, the beds created within the ward should be aborted and similarly if the bed creation query fails, the ward should also be deleted/rolled back. Thus, encapsulating both in a transaction is essential.

How transaction is implemented:
The transaction auto-commit is enabled by default, we disabled it at the beginning of ward creation method to implement transactions. The transaction is committed when both the ward and all its beds are created successfully. If an exception occurs while either of the query fails, the transaction is rolled back in the catch block of exception handling. Also, the transaction auto-commit is enabled post transaction in the method.

Code:

```java
private static void createNewWard(Scanner input) throws SQLException {
   try {
      Connector.setAutoCommit(false);
       int capacity = 0, charges = 0;
       System.out.println("Enter Ward Capacity:");
       capacity = input.nextInt();
      System.out.println("Enter Ward Charges:");
      charges = input.nextInt();
      while(true) {
         System.out.println("Enter Responsible Nurse:");
         int staffID = input.nextInt();
         if(validateNurse(staffID)) {
            Connector.createPreparedStatement(Constants.createWard);
            Connector.setPreparedStatementInt(1, capacity);
            Connector.setPreparedStatementInt(2, charges);
            Connector.setPreparedStatementInt(3, staffID);
            break;
         }
         else
            System.out.println("Nurse does not exist. try again.");
      }
      Connector.executeUpdatePreparedQuery();
      ResultSet rs = Connector.getGeneratedKeys();
      int wardNo = 0;
      if(rs.next())
         wardNo = rs.getInt(1);
      else
         throw new SQLException();
      char alphabet = 'A';
      for(int i = 0; i < capacity; i++) {
         Connector.createPreparedStatement(Constants.createBed);
```

```java
            Connector.setPreparedStatementString(1, ""+alphabet);
            Connector.setPreparedStatementInt(2, wardNo);
            Connector.setPreparedStatementString(3, "Y");
            Connector.executeUpdatePreparedQuery();
            alphabet++;
         }
         Connector.commit();
         System.out.println("Ward "+ wardNo +" added Successfully");
      } catch (SQLException e) {
         Connector.rollback();
         System.out.println("Error occured while processing the data" + e.getMessage());
      }
      Connector.setAutoCommit(true);
   }
```

Use Case 2:
When a patient visits the hospital, a Billing Record is created for him that stores his billing details like fees, payment method etc. Along with it, a Medical Record must also be created at his visit that stores his medical details for the particular treatment he is visiting for. Both Billing Record and Medical Record are associated with each other and have a one to one mapping.

Why implemented using Transaction:
If the Billing Record creation fails, the Medical Record created with it should be aborted and similarly if the Medical Record creation query fails, the Billing Record should also be deleted/rolled back. Thus, encapsulating both in a transaction is essential.

How transaction is implemented:
The transaction auto-commit is enabled by default, we disabled it at the beginning of Billing Record creation method to implement transactions. The transaction is committed when both the Billing Record and its associated Medical Record are created successfully. If an exception occurs while either of the query fails, the transaction is rolled back in the catch block of exception handling. Also, the transaction auto-commit is enabled post transaction in the method.

Code:

```java
private static void createBillingRecord(Scanner input) throws SQLException {
   System.out.println("Do you want to admit the patient(Y/N)?:");
   String admit = input.next();
   boolean bedAvailable = true;
   if(admit.equalsIgnoreCase("Y")) {
      float result = returnWardUsagePercentage();
      if(result < 0.0) {
         throw new SQLException();
      } else if (result < 100.0){
         bedAvailable = true;
      } else
         bedAvailable = false;
   }
   if(!bedAvailable) {
      System.out.println("No empty bed available to check in the patient");
   } else {
      try {
         System.out.println("Enter Patient ID:");
```

```java
        int patientId = input.nextInt();
        System.out.println("Enter the Id of the Responsible Doctor for this treatment: ");
        int docId = input.nextInt();

        Connector.createPreparedStatement(Constants.validatePatient);
        Connector.setPreparedStatementInt(1, patientId);
           ResultSet patRes =  Connector.executePreparedQuery();
           Connector.createPreparedStatement(Constants.validateDoctor);
        Connector.setPreparedStatementInt(1, docId);
           ResultSet docRes =  Connector.executePreparedQuery();
           if(patRes.next() && docRes.next()) {
              Connector.setAutoCommit(false);
            Connector.createPreparedStatement(Constants.createMedicalRecord);
            Connector.setPreparedStatementInt(1, docId);
            Connector.setPreparedStatementInt(2, patientId);
            Connector.executeUpdatePreparedQuery();
            ResultSet rs = Connector.getGeneratedKeys();
            int medId = 0;
            if(rs.next())
               medId = rs.getInt(1);
            else
               throw new SQLException();
            Connector.createPreparedStatement(Constants.createBillingRecord);
            Connector.setPreparedStatementInt(1, Integer.parseInt(User.id));
            Connector.setPreparedStatementInt(2, patientId);
            Connector.setPreparedStatementInt(3, medId);
            System.out.println("Enter Payment Method(Card/Cash/Insurance):");
            String paymentMethod = input.next();
            Connector.setPreparedStatementString(4, paymentMethod);
            if(!paymentMethod.equalsIgnoreCase("cash")) {
               System.out.println("Enter card/insurance details:");
               String temp = input.next();
               Connector.setPreparedStatementString(5, temp);
               System.out.println("Enter billing address:");
               temp = input.next();
               Connector.setPreparedStatementString(8, temp);
            } else {
               Connector.setPreparedStatementString(5, null);
               Connector.setPreparedStatementString(8, null);
            }

            System.out.println("Enter Fees:");
            float fees = input.nextFloat();
            Connector.setPreparedStatementFloat(6, fees);
            System.out.println("Do you want to enter PayeeSSN(optional)? (Y/N):");
            String temp = input.next();
            if(temp.equals("Y")) {
               System.out.println("Enter PayeeSSN:");
               temp = input.next();
               Connector.setPreparedStatementString(7, temp);
            } else {
               Connector.setPreparedStatementString(7, null);
            }
            Connector.executeUpdatePreparedQuery();
            Connector.commit();
            Connector.setAutoCommit(true);
            System.out.println("Billing record created successfully");

         } else {
                System.out.println("Given patient ID or Responsible doctor ID doesn't exist, try again!");
            }
      }
        catch(SQLException e) {
         Connector.rollback();
         Connector.setAutoCommit(true);
         System.out.println("Error occured while creating entries, please check your input data " + e.getMessage());
      }
   }
}
```

High Level Design

Roles:
Mohd Sharique Khan (mkhan8): Software Engineer & Tester, Database Designer/Administrator, API Designer
Richa Dua (rdua2): Software Engineer & Tester, Database Designer/Administrator, Technical Analyst
Siddu Madhure Jayanna (smadhur): Software Engineer & Tester, Database Designer/Administrator, Technical Analyst
Viviniya Alexis Lawrence (palexis): Software Engineer & Tester, Database Designer/Administrator, Technical Analyst

Database Design:
1. Information from the project narrative was extracted to identify the type of data to be recorded in the database, such as Doctor's specialization, Patient's address and hospital Ward etc.
2. The information was divided into major entities such as. Doctor, Patient, Medical Record, which in turn becomes tables. The data that becomes columns were also identified.
3. Primary Key, which is a column or set of columns is chosen for each table.
4. Data relations between different tables were identified and fields like foreign keys were added to clarify the relations.
5. 3NF and BCNF normalizations were applied to check if tables are structured correctly.

Application Design:
The entire java application is conceptualized using module design pattern, thereby separating different modules such as Initializer, Connector, Data models, Validation and Util classes.
1. Data Models: Has all the menu options available for each user, such as Doctor, Registration Staff, Patient and Nurse along with the related API implementations.
2. Initializer (Index/Connector) Class: Used to get the database details and establish connection to the MariaDB.
3. Validation Class: Used to validate the presence of any data in database.
4. Util Class: Helper class with static methods, is a stateless class with bunch of related methods, so they can be reused across the application.
5. User Class: Used to validate all the users of the application against the DB data.