CSE 414 – DATABASE PROJECT REPORT

Fatih Selim YAKAR / 161044054

Lecturer : Dr. Burcu YILMAZ

• USER REQUIREMENTS DESCRIPTION

○ Clarification

This database management system can be used for different purposes. These main purposes were established for the two main users, car customers and employees. And other features have been added to bring the two main users together in a way that suits its purpose.

○ Functional Requirements

▪ Customer

In this database system, it is inherited under the Customer main table; test, maintenance, rental and purchase customers. These customers can make an appointment for their own car or a car already in the showroom and choose the type of appointment. He can also access the appointment's information. The specified customers can learn the price of the work done and the details about the fee as a result of their appointments.

▪ Appointment

In this database system, it is inherited under the Appointment main table; test, maintenance, rental and purchase appointments. These appointments can provide access to the invoice or the car or customers that the appointment belongs to through the appointment. It can also access the employee responsible for the appointment.

▪ Car

In this database system, it is inherited under the Car main table; There are truck, sportscar, normal car, suv and electric vehicle cars. These cars have different features depending on the type. You can access the engine, tire and equipment of the cars from the Car table. There is also access to the showroom where the car is located and to the department where the showroom is located.

▪ Employee

In this database system, it is inherited under the Employee main table; It has technician, manager, sales responsible, reception, leasing responsible employees. These employees have their own unique features and can be linked to a specific appointment/job. In addition, all employees are affiliated to a department, and departments are affiliated to a headquarter.

- ### Non-Functional Requirements

  Since there is a large amount of inheritance in the database system and sql does not support inheritence directly, if an entry of a child table occurs, it must be created in the entry of the parent table it is connected to. This also applies to the deletion process. In addition, the attributes that will be used frequently should be indexed with the indexing system and the access speed should be increased. Again, the view should be made by combining the separate attributes that the user can pull from the database frequently and together. Other than these, checks should be made to prevent incorrect attribute values. In order for this entire system to work from the interface, a well-designed database connection must be made with the JDBC system.

• CAR GALLERY DBMS DESCRIPTION

In this project, the database implementation and interface of a versatile car gallery will be made. First of all, there will be a headquarter section, this section will keep the name of the company, only one manager of the headquarter and he will manage the company. In addition, each department that will have departments under the headquarter will be connected to the main headquarter and the headquarter can have many departments.

In addition, there will be showrooms/multi-purpose rooms under each department, as each showroom will be connected to a department, it will not be necessary to have a showroom in a department. Again, there will be employees under the department. Each employee will have a type and these types will have different features. These employee types will be technician, manager, sales-responsible, reception and leasing responsible.

In the showroom under the aforementioned department, cars can be found, but not necessarily cars. Again, their cars will have different types and features that differentiate these types. These car types will be: truck, sports car, normal car, suv and electric vehicle.

Each car will also have components it owns. These components will be components such as engine, tire, equipment. These components will have no features without the car. So there will be car dependent components. These components can be common to several cars, that is, a certain brand of wheel or engine (as a type) can be used in several cars.

While the interior of the car gallery is like this, if we come to the exterior, first of all, it will be the customer and the customer will have their own different types. These types will be test customer, maintanence customer, leasing customer, saling customer.

This above-mentioned Customer will also be able to make an appointment, and this appointment will have different types and features where these types differ. These appointment types will be: maintanence, leasing, sale and test appointments. There will be those who are interested in the works in these appointments from the employee types depending on the department.

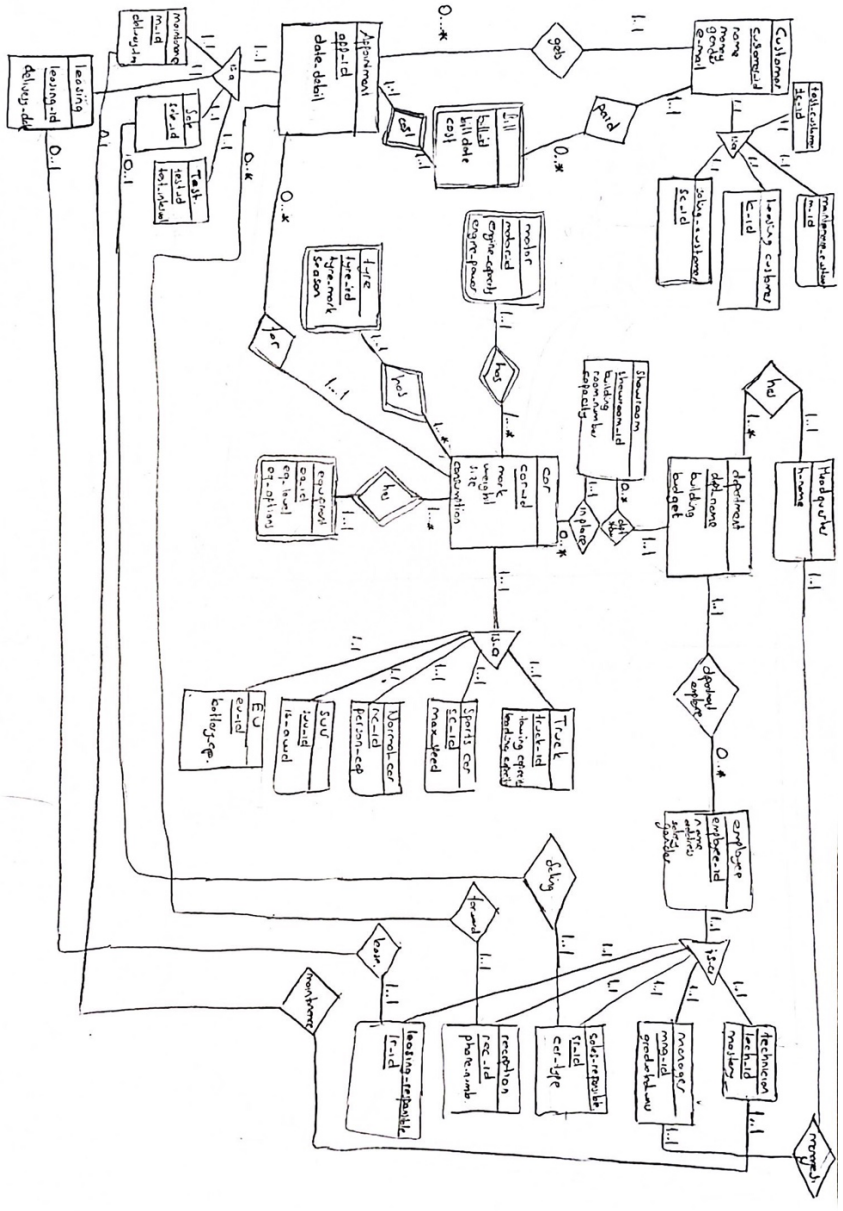
Finally, there will be a bill to the customer for the work done after the appointment. Every bill will have a customer and an appointment, but not every customer need to have a bill.

• ER DIAGRAMS

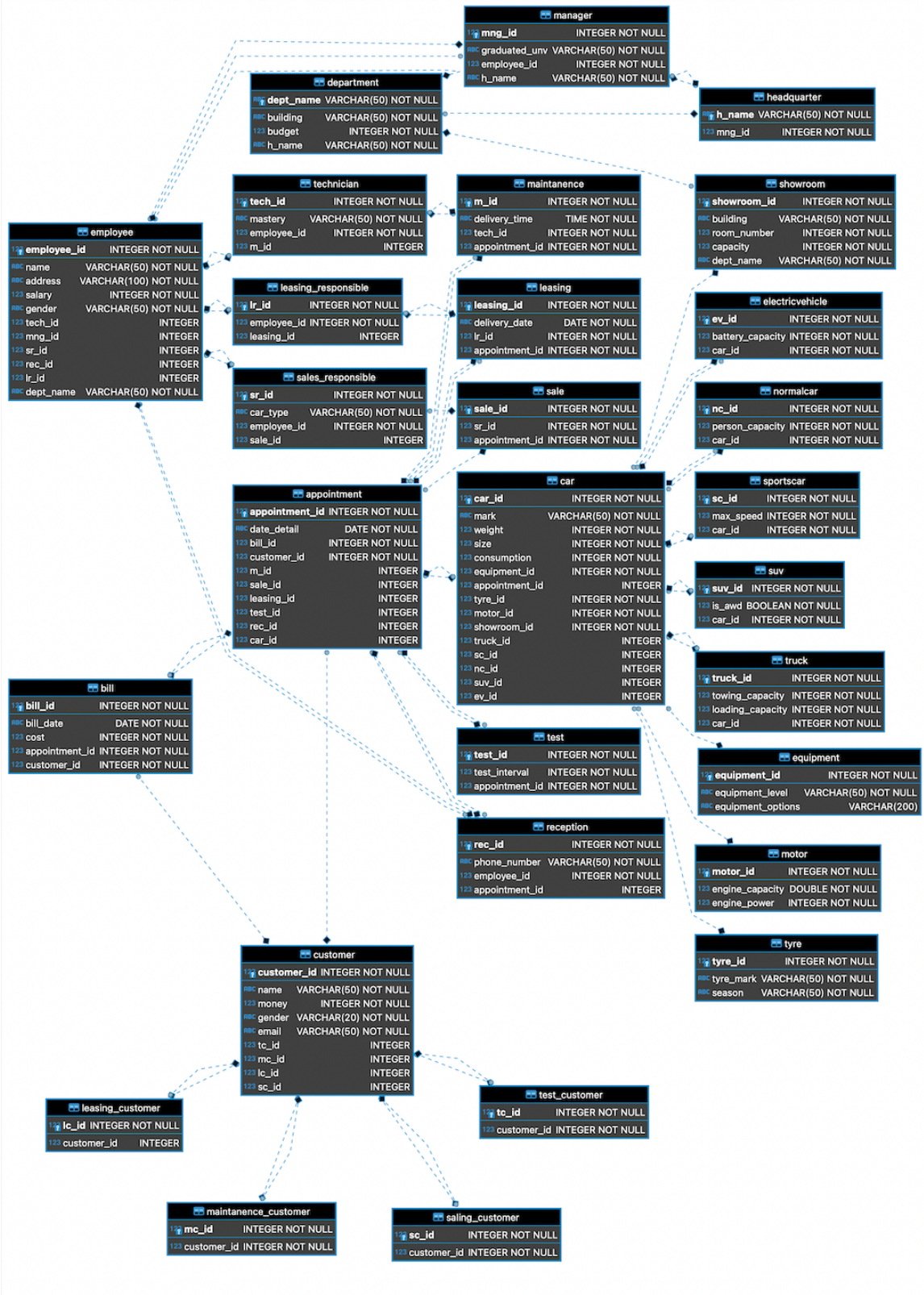
○ Description

A database system was designed to meet the needs defined in the User Requirements Description, Entities and Relations were created. Entities that are related to each other's creation were defined as "weak entities". Entities of the same type were defined as "is-a" relation. Other parts were defined by giving the name relation according to their dependencies.

○ Handmade ER diagram (To The Right To Be Larger)



- **Reverse Engineered ER diagram (Made with DBeaver)**



• FDs, NORMALIZATION AND TABLES

○ BCNF Definition:

A relation scheman R is in BCNF with respect to a set F of functional dependencies if for all functional dependency in F+ of the form

$$\propto \rightarrow \beta$$

Where $\propto \subseteq R$ and $\beta \subseteq R$ at least one of the following holds:

- $\propto \rightarrow \beta$ is trivial ($\beta \subseteq \propto$)
- $\propto$ is superkey for R

○ "Customer" Table

- **Relation Schema**

  Customer(customer_id,name,money,gender,e-mail)

- **Functional Dependencies (Non-Trivial)**

  Customer_id →name, money, gender, e-mail

  e-mail→customer_id, money, gender, e-mail

- **Functional Dependencies (Trivial)**

  customer_id,name,money,gender,e-mail →customer_id

  customer_id,name,money,gender,e-mail →name

  customer_id,name,money,gender,e-mail →money

  customer_id,name,money,gender,e-mail →gender

  customer_id,name,money,gender,e-mail →e-mail

  These can be summarized that:

  customer_id,name,money,gender,e-mail

  →customer_id,name,money,gender,e-mail

  ***Note**: I will use the notation I used to describe the trivial functional dependencies I mentioned above in all other tables. If we continue:*

  customer_id,name,money,gender → customer_id,name,money,gender

  customer_id,name,money,e-mail → customer_id,name,money,e-mail

customer_id,name,gender,e-mail → customer_id,name,gender,e-mail

customer_id,money,gender,e-mail →customer_id,money,gender,e-mail

name,money,gender,e-mail → name,money,gender,e-mail

customer_id,name,money → customer_id,name,money

.

.

money,gender,e-mail → money,gender,e-mail

customer_id,name → customer_id,name

.

.

gender,e-mail →gender, e-mail

customer_id → customer_id

name → name

money → money

gender → gender

e-mail → e-mail

- **Primary Key**

  {{customer_id}}

- **Candidate Keys**

  {{customer_id},{e-mail}}

- **Normal Form**

  BCNF

○ "Test_customer" Table

- **Relation Schema**

  test_customer(tc_id)

- **Functional Dependencies(Non-Trivial)**

  There are not functional dependencies.

- **Functional Dependencies(Trivial)**

  tc_id → tc_id

- **Primary Key**

  {{tc_id}}
- **Candidate Keys**

  {{tc_id}}
- **Normal Form**

  BCNF

○ "Maintanence_customer" Table

- **Relation Schema**

  maintanence_customer(mc_id)
- **Functional Dependencies(Non-Trivial)**

  There are not functional dependencies.
- **Functional Dependencies(Trivial)**

  mc_id → mc_id
- **Primary Key**

  {{mc_id}}
- **Candidate Keys**

  {{mc_id}}
- **Normal Form**

  BCNF

○ "Leasing_customer" Table

- **Relation Schema**

  leasing_customer(lc_id)
- **Functional Dependencies(Non-Trivial)**

  There are not functional dependencies.
- **Functional Dependencies(Trivial)**

  lc_id → lc_id
- **Primary Key**

{{lc_id}}

- **Candidate Keys**

  {{lc_id}}

- **Normal Form**

  BCNF

“Saling_customer” Table

- **Relation Schema**

  saling_customer(sc_id)

- **Functional Dependencies(Non-Trivial)**

  There are not functional dependencies.

- **Functional Dependencies(Trivial)**

  sc_id → sc_id

- **Primary Key**

  {{sc_id}}

- **Candidate Keys**

  {{sc_id}}

- **Normal Form**

  BCNF

“Bill” Table

- **Relation Schema**

  bill (bill_id,bill_date,cost)

- **Functional Dependencies(Non-Trivial)**

  bill_id →bill_date, cost

- **Functional Dependencies(Trivial)**

  bill_id,bill_date,cost → bill_id,bill_date,cost

  bill_id,bill_date→ bill_id,bill_date

  bill_id,cost → bill_id,cost

bill_date,cost → bill_date,cost

bill_id → bill_id

bill_date → bill_date

cost → cost

- **Primary Key**

  {{bill_id}}

- **Candidate Keys**

  {{bill_id}}

- **Normal Form**

  BCNF

- **"Appointment" Table**
  - **Relation Schema**

    appointment (appointment_id,date_detail)

  - **Functional Dependencies(Non-Trivial)**

    appointment_id →date_detail

  - **Functional Dependencies(Non-Trivial)**

    appointment_id,date_detail → appointment_id,date_detail

    appointment_id → appointment_id

    date_detail → date_detail

  - **Primary Key**

    {{appointment_id}}

  - **Candidate Keys**

    {{appointment_id}}

  - **Normal Form**

    BCNF

- **"Maintanence" Table (type of Appointment)**
  - **Relation Schema**

maintanence (m_id,delivery_time)

- **Functional Dependencies(Non-Trivial)**

  m_id →delivery_time

- **Functional Dependencies(Trivial)**

  m_id,delivery_time → m_id,delivery_time

  m_id → m_id

  delivery_time → delivery_time

- **Primary Key**

  {{m_id}}

- **Candidate Keys**

  {{m_id}}

- **Normal Form**

  BCNF

- **"Sale" Table (type of Appointment)**

  - **Relation Schema**

    sale (sale_id)

  - **Functional Dependencies(Non-Trivial)**

    There are not functional dependencies

  - **Functional Dependencies(Trivial)**

    sale_id → sale_id

  - **Primary Key**

    {{sale_id}}

  - **Candidate Keys**

    {{sale_id}}

  - **Normal Form**

    BCNF

- **"Test" Table (type of Appointment)**
  - **Relation Schema**

    test (test_id,test_interval)
  - **Functional Dependencies(Non-Trivial)**

    test_id → test_interval
  - **Functional Dependencies(Trivial)**

    test_id, test_interval → test_id, test_interval

    test_id → test_id

    test_interval → test_interval
  - **Primary Key**

    {{test_id}}
  - **Candidate Keys**

    {{test _id}}
  - **Normal Form**

    BCNF

- **"Leasing" Table (type of Appointment)**
  - **Relation Schema**

    leasing (leasing_id,delivery_date)
  - **Functional Dependencies(Non-Trivial)**

    leasing_id → delivery_date
  - **Functional Dependencies(Trivial)**

    leasing_id,delivery_date → leasing_id,delivery_date

    leasing_id → leasing_id

    delivery_date → delivery_date
  - **Primary Key**

    {{leasing_id}}
  - **Candidate Keys**

    {{leasing_id}}

- **Normal Form**

  BCNF

“Headquarter” Table

- **Relation Schema**

  headquarter (h_name)

- **Functional Dependencies(Non-Trivial)**

  There are not functional dependencies

- **Functional Dependencies(Trivial)**

  h_name → h_name

- **Primary Key**

  {{h_name}}

- **Candidate Keys**

  {{h_name}}

- **Normal Form**

  BCNF

“Department” Table

- **Relation Schema**

  department (dept_name,building,budget)

- **Functional Dependencies(Non-Trivial)**

  dept_name → building, budget

- **Functional Dependencies(Trivial)**

  dept_name,building,budget → dept_name,building,budget

  dept_name,building → dept_name,building

  dept_name, budget → dept_name, budget

  building,budget → building,budget

  dept_name → dept_name

  building → building

budget → budget

- **Primary Key**

  {{dept_name}}

- **Candidate Keys**

  {{dept_name}}

- **Normal Form**

  BCNF

○ "Showroom" Table

- **Relation Schema**

  showroom (showroom_id,building,room_number,capacity)

- **Functional Dependencies(Non-Trivial)**

  showroom_id → building, room_number, capacity

- **Functional Dependencies(Trivial)**

  showroom_id,building,room_number,capacity →
  showroom_id,building,room_number,capacity
  showroom_id,building,room_number →
  showroom_id,building,room_number
  showroom_id,building,capacity → showroom_id,building, capacity
  showroom_id, room_number,capacity → showroom_id, room_number, capacity
  building, room_number, capacity → building, room_number, capacity
  showroom_id,building → showroom_id,building
  .
  .
  room_number,capacity → room_number,capacity
  showroom_id → showroom_id
  building →building
  room_number → room_number
  capacity → capacity

- **Primary Key**

  {{showroom_id}}
- **Candidate Keys**

  {{showroom_id}}
- **Normal Form**

  BCNF

○ "Car" Table

- **Relation Schema**

  car (<u>car_id</u>,mark,weight,size,consumption)
- **Functional Dependencies(Non-Trivial)**

  car_id → mark, weight, size, consumption
- **Functional Dependencies(Trivial)**

  car_id,mark,weight,size,consumption →

  car_id,mark,weight,size,consumption

  car_id,mark,weight,size → car_id,mark,weight,size

  car_id,mark,weight, consumption → car_id,mark,weight, consumption

  car_id,mark, size, consumption → car_id,mark, size, consumption

  car_id, weight, size, consumption → car_id, weight, size, consumption

  mark,weight,size,consumption → mark,weight,size,consumption

  car_id,mark,weight → car_id,mark,weight

  .

  .

  weight,size,consumption → weight,size,consumption

  car_id,mark → car_id,mark

  .

  .

  size, consumption → size, consumption

  car_id → car_id

  mark → mark

weight → weight

size → size

consumption → consumption

- **Primary Key**

  {{car_id}}

- **Candidate Keys**

  {{car_id}}

- **Normal Form**

  BCNF

- **"Truck" Table**
  - **Relation Schema**

    truck (truck_id,towing_capacity,loading_capacity)

  - **Functional Dependencies(Non-Trivial)**

    truck_id → towing_capacity, loading_capacity

  - **Functional Dependencies(Trivial)**

    truck_id,towing_capacity,loading_capacity →

    truck_id,towing_capacity,loading_capacity

    truck_id,towing_capacity → truck_id,towing_capacity

    truck_id,loading_capacity → truck_id,loading_capacity

    towing_capacity,loading_capacity → towing_capacity,loading_capacity

    truck_id → truck_id,

    towing_capacity → towing_capacity

    loading_capacity → loading_capacity

  - **Primary Key**

    {{truck _id}}

  - **Candidate Keys**

    {{truck _id}}

  - **Normal Form**

    BCNF

- **"Sportscar" Table**
  - **Relation Schema**

    sportscar (sc_id,max_speed)
  - **Functional Dependencies(Non-Trivial)**

    sc_id → max_speed
  - **Functional Dependencies(Trivial)**

    sc_id,max_speed → sc_id,max_speed

    sc_id → sc_id

    max_speed → max_speed
  - **Primary Key**

    {{ sc_id }}
  - **Candidate Keys**

    {{ sc_id }}
  - **Normal Form**

    BCNF

- **"Normalcar" Table**
  - **Relation Schema**

    normalcar (nc_id,person_capacity)
  - **Functional Dependencies(Non-Trivial)**

    nc _id → person_capacity
  - **Functional Dependencies(Trivial)**

    nc_id,person_capacity → nc_id,person_capacity

    nc _id → nc _id

    person_capacity → person_capacity
  - **Primary Key**

    {{ nc_id }}
  - **Candidate Keys**

{{ nc_id }}

- **Normal Form**

  BCNF

“Suv” Table

- **Relation Schema**

  suv (suv_id,is_awd)

- **Functional Dependencies(Non-Trivial)**

  suv _id → is_awd

- **Functional Dependencies(Trivial)**

  suv_id,is_awd → suv_id,is_awd

  suv _id → suv _id

  is_awd → is_awd

- **Primary Key**

  {{ suv_id }}

- **Candidate Keys**

  {{ suv_id }}

- **Normal Form**

  BCNF

“Electricvehicle” Table

- **Relation Schema**

  electricvehicle (ev_id,battery_capacity)

- **Functional Dependencies(Non-Trivial)**

  ev _id → battery_capacity

- **Functional Dependencies(Trivial)**

  ev_id, battery_capacity → ev_id, battery_capacity

  ev _id → ev_id

  battery_capacity → battery_capacity

- **Primary Key**

  {{ ev_id }}
- **Candidate Keys**

  {{ ev_id }}
- **Normal Form**

  BCNF

- **"Motor" Table**
  - **Relation Schema**

    motor (motor_id,engine_capacity,engine_power)
  - **Functional Dependencies(Non-Trivial)**

    motor _id → engine_capacity,engine_power
  - **Functional Dependencies(Trivial)**

    motor_id,engine_capacity,engine_power →

    motor_id,engine_capacity,engine_power

    motor_id,engine_capacity → motor_id,engine_capacity

    motor_id, engine_power → motor_id, engine_power

    engine_power, engine_capacity → engine_power, engine_capacity

    motor_id → motor_id

    engine_capacity → engine_capacity
  - **Primary Key**

    {{ motor_id }}
  - **Candidate Keys**

    {{ motor_id }}
  - **Normal Form**

    BCNF

- **"Tyre" Table**
  - **Relation Schema**

tyre (tyre_id,tyre_mark,season)

- **Functional Dependencies(Non-Trivial)**

  tyre _id → tyre_mark,season

- **Functional Dependencies(Trivial)**

  tyre_id,tyre_mark,season → tyre_id,tyre_mark,season

  tyre_id,tyre_mark → tyre_id,tyre_mark

  tyre_id, season → tyre_id, season

  season,tyre_mark → season,tyre_mark

  tyre_id → tyre_id

  tyre_mark → tyre_mark

  season → season

- **Primary Key**

  {{ tyre _id }}

- **Candidate Keys**

  {{ tyre _id }}

- **Normal Form**

  BCNF

○ "Equipment" Table

- **Relation Schema**

  equipment (equipment_id,equipment_level,equipment_options)

- **Functional Dependencies(Non-Trivial)**

  equipment _id → equipment_level,equipment_options

- **Functional Dependencies(Trivial)**

  equipment_id,equipment_level,equipment_options →

  equipment_id,equipment_level,equipment_options

  equipment_id,equipment_level → equipment_id,equipment_level

  equipment_id,equipment_options → equipment_id,equipment_options

  equipment_level,equipment_options →

  equipment_level,equipment_options

equipment_id → equipment_id

equipment_level → equipment_level

equipment_options → equipment_options

- **Primary Key**

  {{ equipment _id }}

- **Candidate Keys**

  {{ equipment _id }}

- **Normal Form**

  BCNF

○ "Employee" Table

- **Relation Schema**

  employee(employee_id, name,address,salary,gender)

- **Functional Dependencies(Non-Trivial)**

  employee_id → name,address,salary,gender

- **Functional Dependencies(Trivial)**

  employee_id, name,address,salary,gender → employee_id, name,address,salary,gender

  employee_id, name,address,salary → employee_id, name,address,salary

  employee_id, name,address, gender → employee_id, name,address, gender

  employee_id, name,salary, gender → employee_id, name,salary, gender

  employee_id,address,salary,gender → employee_id,address,salary,gender

  name,address,salary,gender → name,address,salary,gender

  employee_id, name,addres → employee_id, name,addres

  .

  .

address,salary,gender → address,salary,gender

employee_id, name → employee_id, name

.

.

salary,gender → salary,gender

employee_id → employee_id

name → name

address → address

salary → salary

gender → gender

- **Primary Key**

  {{ employee_id }}

- **Candidate Keys**

  {{ employee_id }}

- **Normal Form**

  BCNF

○ "Technician" Table

- **Relation Schema**

  technician(<u>tech_id</u>, mastery)

- **Functional Dependencies(Non-Trivial)**

  tech_id → mastery

- **Functional Dependencies(Trivial)**

  tech_id, mastery → tech_id, mastery

  tech_id → tech_id

  mastery → mastery

- **Primary Key**

  {{ tech_id }}

- **Candidate Keys**

  {{ tech_id }}

- **Normal Form**

  BCNF

○ "Manager" Table

- **Relation Schema**

  manager(<u>mng_id</u>, graduated_university)
- **Functional Dependencies(Non-Trivial)**

  mng_id → graduated_university
- **Functional Dependencies(Trivial)**

  mng_id, graduated_university → mng_id, graduated_university

  mng_id → mng_id

  graduated_university → graduated_university
- **Primary Key**

  {{ mng_id }}
- **Candidate Keys**

  {{ mng_id }}
- **Normal Form**

  BCNF

○ "Sales_responsible" Table

- **Relation Schema**

  Sales_responsible(<u>sr_id</u>, car_type)
- **Functional Dependencies(Non-Trivial)**

  sr_id → car_type
- **Functional Dependencies(Trivial)**

  sr_id, car_type → sr_id, car_type

  sr_id → sr_id

  car_type → car_type
- **Primary Key**

{{ sr_id }}

- **Candidate Keys**

  {{ sr_id }}

- **Normal Form**

  BCNF

- **"Reception" Table**

  - **Relation Schema**

    reception(rec_id, phone_number)

  - **Functional Dependencies(Non-Trivial)**

    rec_id → phone_number

    phone_number →rec_id

  - **Functional Dependencies(Trivial)**

    rec_id, phone_number → rec_id, phone_number

    rec_id → rec_id

    phone_number → phone_number

  - **Primary Key**

    {{ sr_id }}

  - **Candidate Keys**

    {{ sr_id, phone_number }}

  - **Normal Form**

    BCNF

- **"Leasing_responsible" Table**

  - **Relation Schema**

    leasing_responsible(lr_id)

  - **Functional Dependencies(Non-Trivial)**

    There are not functional dependencies

  - **Functional Dependencies(Trivial)**

lr_id → lr_id

- **Primary Key**

  {{ lr_id }}

- **Candidate Keys**

  {{ lr_id }}

- **Normal Form**

  BCNF

- # TABLE IMPLEMENTATIONS

All this was done using **sqlite and sqlite studio** with the implementation to be mentioned. SQLite is an in-process library that implements a self-contained, serverless, zero-configuration, transactional SQL database engine. The code for SQLite is in the public domain and is thus free for use for any purpose, commercial or private. SQLite is the most widely deployed database in the world with more applications than we can count, including several high-profile projects.

```
CREATE TABLE IF NOT EXISTS technician(
    tech_id integer primary key autoincrement not null,
    mastery varchar(50) not null,

    employee_id integer not null,
    m_id integer,
    foreign key (employee_id) references employee(employee_id),
    foreign key (m_id) references maintanence(m_id)
);
CREATE INDEX index_technician ON technician(mastery);

CREATE TABLE IF NOT EXISTS manager(
    mng_id integer primary key autoincrement not null,
    graduated_unv varchar(50) not null,

    employee_id integer not null,
    h_name varchar(50) not null,
    foreign key (employee_id) references employee(employee_id),
    foreign key (h_name) references headquarter(h_name)
);
CREATE INDEX index_manager ON manager(graduated_unv);

CREATE TABLE IF NOT EXISTS sales_responsible(
    sr_id integer primary key autoincrement not null,
    car_type varchar(50) not null,

    employee_id integer not null,
```

```
    sale_id integer,
    foreign key (employee_id) references employee(employee_id),
    foreign key (sale_id) references sale(sale_id)
);
CREATE INDEX index_sales_reponsible ON sales_responsible(car_type);

CREATE TABLE IF NOT EXISTS reception(
    rec_id integer primary key autoincrement not null,
    phone_number varchar(50) not null,

    employee_id integer not null,
    appointment_id integer,
    foreign key (employee_id) references employee(employee_id),
    foreign key (appointment_id) references appointment(appointment_id)
);
CREATE UNIQUE INDEX index_reception ON reception(phone_number);

CREATE TABLE IF NOT EXISTS leasing_responsible(
    lr_id integer primary key autoincrement not null,

    employee_id integer not null,
    leasing_id integer,
    foreign key (employee_id) references employee(employee_id),
    foreign key (leasing_id) references leasing(leasing_id)
);


CREATE TABLE IF NOT EXISTS showroom(
    showroom_id integer primary key autoincrement not null,
    building varchar(50) not null,
    room_number integer not null,
    capacity integer not null,

    dept_name varchar(50) not null,
    foreign key (dept_name) references department(dept_name)
);

CREATE TABLE IF NOT EXISTS employee(
    employee_id integer primary key autoincrement not null,
    name varchar(50) not null,
    address varchar(100) not null,
    salary integer not null,
    gender varchar(50) not null,

    tech_id integer,
    mng_id integer,
    sr_id integer,
    rec_id integer,
```

```
    lr_id integer,
    dept_name varchar(50) not null,

    foreign key (tech_id) references technician(tech_id),
    foreign key (mng_id) references manager(mng_id),
    foreign key (sr_id) references sales_responsible(sr_id),
    foreign key (rec_id) references reception(rec_id),
    foreign key (lr_id) references leasing_responsible(lr_id),
    foreign key (dept_name) references department(dept_name)
);
CREATE INDEX index_employee ON employee(name,address,salary,gender);

CREATE TABLE IF NOT EXISTS department(
    dept_name varchar(50) primary key not null,
    building varchar(50) not null,
    budget integer not null,

    h_name varchar(50) not null,
    foreign key (h_name) references headquarter(h_name)
);
CREATE INDEX index_department ON department(building,budget,dept_name);

CREATE TABLE IF NOT EXISTS headquarter(
    h_name varchar(50) primary key not null,

    mng_id integer not null,
    foreign key (mng_id) references manager(mng_id)
);

CREATE TABLE IF NOT EXISTS equipment(
    equipment_id integer primary key autoincrement not null,
    equipment_level varchar(50) not null,
    equipment_options varchar(200)
);
CREATE INDEX index_equipment ON equipment(equipment_level,equipment_options);

CREATE TABLE IF NOT EXISTS tyre(
    tyre_id integer primary key autoincrement not null,
    tyre_mark varchar(50) not null,
    season varchar(50) not null
);
CREATE INDEX index_tyre ON tyre(tyre_mark,season);

CREATE TABLE IF NOT EXISTS motor(
    motor_id integer primary key autoincrement not null,
    engine_capacity double not null,
    engine_power integer not null
);
```

```
CREATE INDEX index_motor ON motor(engine_capacity,engine_power);

CREATE TABLE IF NOT EXISTS truck(
    truck_id integer primary key autoincrement not null,
    towing_capacity integer not null,
    loading_capacity integer not null,

    car_id integer not null,
    foreign key (car_id) references car(car_id)
);
CREATE INDEX index_truck ON truck(towing_capacity,loading_capacity);

CREATE TABLE IF NOT EXISTS sportscar(
    sc_id integer primary key autoincrement not null,
    max_speed integer not null,

    car_id integer not null,
    foreign key (car_id) references car(car_id)
);
CREATE INDEX index_sportscar ON sportscar(max_speed);

CREATE TABLE IF NOT EXISTS normalcar(
    nc_id integer primary key autoincrement not null,
    person_capacity integer not null,

    car_id integer not null,
    foreign key (car_id) references car(car_id)
);
CREATE INDEX index_normalcar ON normalcar(person_capacity);

CREATE TABLE IF NOT EXISTS suv(
    suv_id integer primary key autoincrement not null,
    is_awd boolean not null,

    car_id integer not null,
    foreign key (car_id) references car(car_id)
);
CREATE INDEX index_suv ON suv(is_awd);

CREATE TABLE IF NOT EXISTS electricvehicle(
    ev_id integer primary key autoincrement not null,
    battery_capacity integer not null,

    car_id integer not null,
    foreign key (car_id) references car(car_id)
);
CREATE INDEX index_ev ON electricvehicle(battery_capacity);
```

```
CREATE TABLE IF NOT EXISTS maintanence(
    m_id integer primary key autoincrement not null,
    delivery_time time not null,

    tech_id integer not null,
    appointment_id integer not null,
    foreign key (tech_id) references technician(tech_id),
    foreign key (appointment_id) references appointment(appointment_id)

);
CREATE INDEX index_maintanence ON maintanence(delivery_time);

CREATE TABLE IF NOT EXISTS sale(
    sale_id integer primary key autoincrement not null,

    sr_id integer not null,
    appointment_id integer not null,
    foreign key (sr_id) references sale_responsible(sr_id),
    foreign key (appointment_id) references appointment(appointment_id)
);

CREATE TABLE IF NOT EXISTS leasing(
    leasing_id integer primary key autoincrement not null,
    delivery_date date not null,

    lr_id integer not null,
    appointment_id integer not null,
    foreign key (lr_id) references leasing_responsible(lr_id),
    foreign key (appointment_id) references appointment(appointment_id)
);
CREATE INDEX index_leasing ON leasing(delivery_date);

CREATE TABLE IF NOT EXISTS test(
    test_id integer primary key autoincrement not null,
    test_interval integer not null,

    appointment_id integer not null,
    foreign key (appointment_id) references appointment(appointment_id)
);
CREATE INDEX index_test ON test(test_interval);

CREATE TABLE IF NOT EXISTS appointment(
    appointment_id integer primary key autoincrement not null,
    date_detail date not null,

    bill_id integer not null,
    customer_id integer not null,
    m_id integer,
```

```
    sale_id integer,
    leasing_id integer,
    test_id integer,
    rec_id integer,
    car_id integer,
    foreign key (bill_id) references bill(bill_id),
    foreign key (customer_id) references customer(customer_id),
    foreign key (m_id) references maintanence(m_id),
    foreign key (sale_id) references sale(sale_id),
    foreign key (leasing_id) references leasing(leasing_id),
    foreign key (test_id) references test(test_id),
    foreign key (rec_id) references reception(rec_id),
    foreign key (car_id) references car(car_id)
);
CREATE INDEX index_appointment ON appointment(date_detail);

CREATE TABLE IF NOT EXISTS bill(
    bill_id integer primary key autoincrement not null,
    bill_date date not null,
    cost integer not null,

    appointment_id integer not null,
    customer_id integer not null,
    foreign key (appointment_id) references appointment(appointment_id)
    foreign key (customer_id) references customer(customer_id)
);
CREATE INDEX index_bill ON bill(bill_date,cost);

CREATE TABLE IF NOT EXISTS test_customer(
    tc_id integer primary key autoincrement not null,

    customer_id integer not null,
    foreign key (customer_id) references customer(customer_id)
);

CREATE TABLE IF NOT EXISTS maintanence_customer(
    mc_id integer primary key autoincrement not null,

    customer_id integer not null,
    foreign key (customer_id) references customer(customer_id)
);

CREATE TABLE IF NOT EXISTS leasing_customer(
    lc_id integer primary key autoincrement not null,

    customer_id integer,
    foreign key (customer_id) references customer(customer_id) on delete cascade
);
```

```
CREATE TABLE IF NOT EXISTS saling_customer(
    sc_id integer primary key autoincrement not null,

    customer_id integer not null,
    foreign key (customer_id) references customer(customer_id)
);

CREATE TABLE IF NOT EXISTS customer(
    customer_id integer primary key autoincrement not null,
    name varchar(50) not null,
    money integer not null,
    gender varchar(20) not null,
    email varchar(50) not null,

    tc_id integer,
    mc_id integer,
    lc_id integer,
    sc_id integer,
    foreign key (tc_id) references test_customer(tc_id),
    foreign key (mc_id) references maintanence_customer(mc_id),
    foreign key (lc_id) references leasing_customer(lc_id) on delete cascade,
    foreign key (sc_id) references saling_customer(sc_id)
);
CREATE INDEX index_customer ON customer(name,money,gender,email);


CREATE TABLE IF NOT EXISTS car(
    car_id integer primary key autoincrement not null,
    mark varchar(50) not null,
    weight integer not null,
    size integer not null,
    consumption integer not null,

    equipment_id integer not null,
    appointment_id integer,
    tyre_id integer not null,
    motor_id integer not null,
    showroom_id integer not null,
    truck_id integer,
    sc_id integer,
    nc_id integer,
    suv_id integer,
    ev_id integer,

    foreign key (equipment_id) references equipment(equipment_id),
    foreign key (appointment_id) references appointment(appointment_id),
    foreign key (tyre_id) references tyre(tyre_id),
```

```
    foreign key (motor_id) references motor(motor_id),
    foreign key (showroom_id) references showroom(showroom_id),
    foreign key (truck_id) references truck(truck_id),
    foreign key (sc_id) references sportscar(sc_id),
    foreign key (nc_id) references normalcar(nc_id),
    foreign key (suv_id) references suv(suv_id),
    foreign key (ev_id) references electricvehicle(ev_id)

);
CREATE INDEX index_car ON car(mark,weight,size,consumption);
```

- **TRIGGERS**

In this section, I will talk about the triggers that I have set up for different purposes in a table-by-table format. In total, 56 triggers were created.

- **Customer Table's Triggers**
  - **Validate Trigger**

    This trigger is written to check the ranges of the instance before adding a new instance to the customer table. The reason for using this instead of the check keyword is to clearly learn the error that occurs in any wrong situation.

```
CREATE TRIGGER IF NOT EXISTS
validate_attribute_range_before_insert_customer
   BEFORE INSERT ON customer
BEGIN
   SELECT
      CASE
          WHEN NEW.email NOT LIKE '%_@__%.__%' THEN RAISE (ABORT,'Invalid
email address')
          WHEN NEW.money<0 THEN RAISE (ABORT,'Invalid money amount')
          WHEN (NEW.gender not in ("Female","Male")) THEN RAISE
(ABORT,'Invalid gender type')
       END;
END;
```

- **Test Customer Table's Triggers**
  - **Insert Trigger**

    Since the test customer table is a customer type, a customer and test customer pair should be formed, the foreign keys of which show each other due to the inheritance relationship during creation. Therefore, when the test customer is created, a trigger is written that creates a customer. First, the customer was created and a test_customer was connected with

the customer's foreign key. Also, it was provided to connect the customer in the newly created test_customer.

```
CREATE TRIGGER IF NOT EXISTS add_customer_after_add_test_customer
   AFTER INSERT ON test_customer
BEGIN
    INSERT INTO customer (name, money, gender, email, tc_id,
mc_id,lc_id,sc_id) VALUES
("",0,"Female","none@none.com",NEW.tc_id,null,null,null);
    UPDATE test_customer SET

    customer_id = CASE
         WHEN (SELECT seq FROM sqlite_sequence WHERE name="customer")>0
THEN (SELECT seq FROM sqlite_sequence WHERE name="customer")+1
         ELSE 1
    END

    WHERE tc_id=NEW.tc_id;
END;
```

- **Delete Trigger**

  Since the test customer table is a type of customer, when the test customer is deleted due to the inheritance relationship during the deletion, the row in the customer table that connects to the test customer must be deleted. This was also done in the trigger below.

```
CREATE TRIGGER IF NOT EXISTS delete_customer_after_delete_test_customer
   AFTER DELETE ON test_customer
BEGIN
    Delete from customer where customer_id=OLD.customer_id;
END;
```

○ Maintanence Customer Table's Triggers

- **Insert Trigger**

  Since the maintanence customer table is a customer type, a customer and maintanence customer pair should be formed, the foreign keys of which show each other due to the inheritance relationship during creation. Therefore, when the maintanence customer is created, a trigger is written that creates a customer. First, the customer was created and a maintanence_customer was connected with the customer's foreign key. Also, it was provided to connect the customer in the newly created maintanence_customer.

```
CREATE TRIGGER IF NOT EXISTS add_customer_after_add_maintanence_customer
   AFTER INSERT ON maintanence_customer
BEGIN
```

```
    INSERT INTO customer (name, money, gender, email, tc_id,
mc_id,lc_id,sc_id) VALUES
("",0,"Female","none@none.com",null,NEW.mc_id,null,null);
    UPDATE maintanence_customer SET

    customer_id = CASE
        WHEN (SELECT seq FROM sqlite_sequence WHERE name="customer")>0
THEN (SELECT seq FROM sqlite_sequence WHERE name="customer")+1
        ELSE 1
    END

    WHERE mc_id=NEW.mc_id;
END;
```

- **Delete Trigger**

  Since the maintanence customer table is a type of customer, when the maintanence customer is deleted due to the inheritance relationship during the deletion, the row in the customer table that connects to the maintanence customer must be deleted. This was also done in the trigger below.

```
CREATE TRIGGER IF NOT EXISTS
delete_customer_after_delete_maintanence_customer
   AFTER DELETE ON maintanence_customer
BEGIN
    Delete from customer where customer_id=OLD.customer_id;
END;
```

Leasing Customer Table's Triggers

- **Insert Trigger**

  Since the leasing customer table is a customer type, a customer and leasing customer pair should be formed, the foreign keys of which show each other due to the inheritance relationship during creation. Therefore, when the leasing customer is created, a trigger is written that creates a customer. First, the customer was created and a leasing_customer was connected with the customer's foreign key. Also, it was provided to connect the customer in the newly created leasing_customer.

```
CREATE TRIGGER IF NOT EXISTS add_customer_after_add_leasing_customer
   AFTER INSERT ON leasing_customer
BEGIN
    INSERT INTO customer (name, money, gender, email, tc_id,
mc_id,lc_id,sc_id) VALUES
("",0,"Female","none@none.com",null,null,NEW.lc_id,null);
    UPDATE leasing_customer SET
```

```
    customer_id = CASE
         WHEN (SELECT seq FROM sqlite_sequence WHERE name="customer")>0
THEN (SELECT seq FROM sqlite_sequence WHERE name="customer")+1
         ELSE 1
    END

    WHERE lc_id=NEW.lc_id;
END;
```

- **Delete Trigger**

  Since the leasing customer table is a type of customer, when the leasing customer is deleted due to the inheritance relationship during the deletion, the row in the customer table that connects to the leasing customer must be deleted. This was also done in the trigger below.

```
CREATE TRIGGER IF NOT EXISTS
delete_customer_after_delete_leasing_customer
   AFTER DELETE ON leasing_customer
BEGIN
    Delete from customer where customer_id=OLD.customer_id;
END;
```

○ Saling Customer Table's Triggers

- **Insert Trigger**

  Since the saling customer table is a customer type, a customer and saling customer pair should be formed, the foreign keys of which show each other due to the inheritance relationship during creation. Therefore, when the saling customer is created, a trigger is written that creates a customer. First, the customer was created and a saling _customer was connected with the customer's foreign key. Also, it was provided to connect the customer in the newly created saling _customer.

```
CREATE TRIGGER IF NOT EXISTS add_customer_after_add_saling_customer
   AFTER INSERT ON saling_customer
BEGIN
    INSERT INTO customer (name, money, gender, email, tc_id,
mc_id,lc_id,sc_id) VALUES
("",0,"Female","none@none.com",null,null,null,NEW.sc_id);
    UPDATE saling_customer SET

    customer_id = CASE
         WHEN (SELECT seq FROM sqlite_sequence WHERE name="customer")>0
THEN (SELECT seq FROM sqlite_sequence WHERE name="customer")+1
         ELSE 1
    END
```

```
    WHERE sc_id=NEW.sc_id;
END;
```

- **Delete Trigger**

  Since the saling customer table is a type of customer, when the saling customer is deleted due to the inheritance relationship during the deletion, the row in the customer table that connects to the saling customer must be deleted. This was also done in the trigger below.

```
CREATE TRIGGER IF NOT EXISTS delete_customer_after_delete_saling_customer
   AFTER DELETE ON saling_customer
BEGIN
    Delete from customer where customer_id=OLD.customer_id;
END;
```

Appointment Table's Triggers

- **Insert Trigger**

  In case of appointment, a bill must be created for the work done to be paid. The resulting bill must be connected to an appointment and a foreign key. To do this, the following trigger was written.

```
CREATE TRIGGER IF NOT EXISTS add_bill_after_add_appointment
   AFTER INSERT ON appointment
BEGIN
    INSERT INTO bill (bill_date, cost, appointment_id, customer_id)
VALUES ("2012-12-12 00:00:00.000",0,NEW.appointment_id,NEW.customer_id);

    UPDATE appointment

    SET bill_id = CASE
         WHEN (SELECT seq FROM sqlite_sequence WHERE name="bill")>0 THEN
(SELECT seq FROM sqlite_sequence WHERE name="bill")+1
         ELSE 1
    END

    WHERE appointment_id=NEW.appointment_id;
END;
```

- **Delete Trigger**

  Since the bill has no meaning on its own, if the appointment is deleted, the bill must be deleted as well. We provide this with the following trigger.

```
CREATE TRIGGER IF NOT EXISTS delete_bill_after_delete_appointment
   AFTER DELETE ON appointment
BEGIN
    Delete from bill where bill_id=OLD.bill_id;
```

```
END;
```

- **Update Trigger**

  In case the customer_id, which is the foreign key in the appointment table, changes, the customer_id shown by the bill to which the appointment is attached should change because it should show the same customers. For this, the following trigger was written.

```
CREATE TRIGGER IF NOT EXISTS update_customer_id_after_update_appointment
AFTER
    UPDATE OF customer_id ON appointment
BEGIN
   UPDATE bill
   SET customer_id=NEW.customer_id
   WHERE appointment_id=NEW.appointment_id;
END;
```

Since the foreign key car_id and the appointment in the appointment table contain a foreign key that points to each other, in case this key is updated, the car_id showing it opposite should show itself and the old one should be null. This was done with the following trigger.

```
CREATE TRIGGER IF NOT EXISTS update_appointment_id_after_update_car_id
AFTER
    UPDATE OF car_id ON appointment
BEGIN
   UPDATE car
   SET appointment_id=NEW.appointment_id
   WHERE car_id=NEW.car_id;

   UPDATE car
   SET appointment_id=null
   WHERE car_id=OLD.car_id AND OLD.appointment_id<>NEW.appointment_id;
END;
```

Bill Table's Triggers

- **Update Trigger**

  In case the customer_id, which is the foreign key in the bill table, changes, the customer_id shown by the appointment to which the bill is attached should change because it should show the same customers. For this, the following trigger was written.

```
CREATE TRIGGER IF NOT EXISTS update_customer_id_after_update_bill AFTER
    UPDATE OF customer_id ON bill
BEGIN
   UPDATE appointment
```

```
   SET customer_id=NEW.customer_id
   WHERE bill_id=NEW.bill_id;
END;
```

- **Validate Trigger**

  This trigger is written to check the ranges of the instance before adding a new instance to the bill table. The reason for using this instead of the check keyword is to clearly learn the error that occurs in any wrong situation.

```
CREATE TRIGGER IF NOT EXISTS validate_attribute_range_before_insert_bill
   BEFORE INSERT ON bill
BEGIN
   SELECT
      CASE
          WHEN NEW.cost<0 THEN RAISE (ABORT,'Invalid cost amount')
       END;
END;
```

Maintanence Table's Triggers

- **Insert Trigger**

  Since the maintanence table is a appointment type, a appointment and maintanence pair should be formed, the foreign keys of which show each other due to the inheritance relationship during creation. Therefore, when maintanence is created, a trigger is written that creates a appointment. First, the appointment was created and a maintanence was connected with the appointment foreign key. Also, it was provided to connect the appointment in the newly created maintanence

```
CREATE TRIGGER IF NOT EXISTS add_appointment_after_add_maintanence
   AFTER INSERT ON maintanence
BEGIN
    INSERT INTO appointment (date_detail, bill_id, customer_id, m_id,
sale_id, leasing_id,test_id,rec_id,car_id) VALUES ("2012-12-12
00:00:00.000",0,0,NEW.m_id,null,null,null,null,null);
    UPDATE maintanence SET

    appointment_id = CASE
         WHEN (SELECT seq FROM sqlite_sequence WHERE
name="appointment")>0 THEN (SELECT seq FROM sqlite_sequence WHERE
name="appointment")+1
         ELSE 1
    END

    WHERE m_id=NEW.m_id;
```

```
END;
```

- **Delete Trigger**

  Since the maintanence table is a type of appointment, when the maintanence is deleted due to the inheritance relationship during the deletion, the row in the appointment table that connects to the maintanence must be deleted. This was also done in the trigger below.

```
CREATE TRIGGER IF NOT EXISTS delete_appointment_after_delete_maintanence
   AFTER DELETE ON maintanence
BEGIN
    Delete from appointment where appointment_id=OLD.appointment_id;
END;
```

○ Leasing Table's Triggers

- **Insert Trigger**

  Since the leasing table is a appointment type, a appointment and leasing pair should be formed, the foreign keys of which show each other due to the inheritance relationship during creation. Therefore, when leasing is created, a trigger is written that creates a appointment. First, the appointment was created and a leasing was connected with the appointment foreign key. Also, it was provided to connect the appointment in the newly created leasing

```
CREATE TRIGGER IF NOT EXISTS add_appointment_after_add_leasing
   AFTER INSERT ON leasing
BEGIN
    INSERT INTO appointment (date_detail, bill_id, customer_id, m_id,
sale_id, leasing_id,test_id,rec_id,car_id) VALUES ("2012-12-12
00:00:00.000",0,0,null,null,NEW.leasing_id,null,null,null);
    UPDATE leasing SET

    appointment_id = CASE
         WHEN (SELECT seq FROM sqlite_sequence WHERE
name="appointment")>0 THEN (SELECT seq FROM sqlite_sequence WHERE
name="appointment")+1
         ELSE 1
    END

    WHERE leasing_id=NEW.leasing_id;
END;
```

- **Delete Trigger**

  Since the leasing table is a type of appointment, when the leasing is deleted due to the inheritance relationship during the deletion, the row in

the appointment table that connects to the leasing must be deleted. This was also done in the trigger below.

```
CREATE TRIGGER IF NOT EXISTS delete_appointment_after_delete_leasing
   AFTER DELETE ON leasing
BEGIN
    Delete from appointment where appointment_id=OLD.appointment_id;
END;
```

- **Sale Table's Triggers**
  - **Insert Trigger**

    Since the sale table is a appointment type, a appointment and sale pair should be formed, the foreign keys of which show each other due to the inheritance relationship during creation. Therefore, when sale is created, a trigger is written that creates a appointment. First, the appointment was created and a sale was connected with the appointment foreign key. Also, it was provided to connect the appointment in the newly created sale

```
CREATE TRIGGER IF NOT EXISTS add_appointment_after_add_sale
   AFTER INSERT ON sale
BEGIN
    INSERT INTO appointment (date_detail, bill_id, customer_id, m_id,
sale_id, leasing_id,test_id,rec_id,car_id) VALUES ("2012-12-12
00:00:00.000",0,0,null,NEW.sale_id,null,null,null,null);
    UPDATE sale SET

    appointment_id = CASE
         WHEN (SELECT seq FROM sqlite_sequence WHERE
name="appointment")>0 THEN (SELECT seq FROM sqlite_sequence WHERE
name="appointment")+1
         ELSE 1
    END

    WHERE sale_id=NEW.sale_id;
END;
```

  - **Delete Trigger**

    Since the sale table is a type of appointment, when the sale is deleted due to the inheritance relationship during the deletion, the row in the appointment table that connects to the sale must be deleted. This was also done in the trigger below.

```
CREATE TRIGGER IF NOT EXISTS delete_appointment_after_delete_sale
   AFTER DELETE ON sale
BEGIN
    Delete from appointment where appointment_id=OLD.appointment_id;
```

```
END;
```

- ## Test Table's Triggers
  - ### Validate Trigger

    This trigger is written to check the ranges of the instance before adding a new instance to the test table. The reason for using this instead of the check keyword is to clearly learn the error that occurs in any wrong situation.

```
CREATE TRIGGER IF NOT EXISTS validate_attribute_range_before_insert_test
   BEFORE INSERT ON test
BEGIN
   SELECT
      CASE
          WHEN NEW.test_interval<0 THEN RAISE (ABORT,'Invalid test
interval amount')
       END;
END;
```

  - ### Insert Trigger

    Since the test table is a appointment type, a appointment and test pair should be formed, the foreign keys of which show each other due to the inheritance relationship during creation. Therefore, when test is created, a trigger is written that creates a appointment. First, the appointment was created and a test was connected with the appointment foreign key. Also, it was provided to connect the appointment in the newly created test

```
CREATE TRIGGER IF NOT EXISTS add_appointment_after_add_test
   AFTER INSERT ON test
BEGIN
    INSERT INTO appointment (date_detail, bill_id, customer_id, m_id,
sale_id, leasing_id,test_id,rec_id,car_id) VALUES ("2012-12-12
00:00:00.000",0,0,null,null,null,NEW.test_id,null,null);
    UPDATE test SET

    appointment_id = CASE
         WHEN (SELECT seq FROM sqlite_sequence WHERE
name="appointment")>0 THEN (SELECT seq FROM sqlite_sequence WHERE
name="appointment")+1
         ELSE 1
    END

    WHERE test_id=NEW.test_id;
END;
```

- **Delete Trigger**

  Since the test table is a type of appointment, when the test is deleted due to the inheritance relationship during the deletion, the row in the appointment table that connects to the test must be deleted. This was also done in the trigger below.

```
CREATE TRIGGER IF NOT EXISTS delete_appointment_after_delete_test
   AFTER DELETE ON test
BEGIN
    Delete from appointment where appointment_id=OLD.appointment_id;
END;
```

Car Table's Triggers

- **Validate Trigger**

  This trigger is written to check the ranges of the instance before adding a new instance to the car table. The reason for using this instead of the check keyword is to clearly learn the error that occurs in any wrong situation.

```
CREATE TRIGGER IF NOT EXISTS validate_attribute_range_before_insert_car
   BEFORE INSERT ON car
BEGIN
   SELECT
      CASE
          WHEN NEW.weight<0 THEN RAISE (ABORT,'Invalid weight amount')
          WHEN NEW.size<0 THEN RAISE (ABORT,'Invalid size amount')
          WHEN NEW.consumption<0 THEN RAISE (ABORT,'Invalid consumption
amount')
       END;
END;
```

- **Update Trigger**

  Since the foreign key car_id and the appointment in the appointment table contain a foreign key that points to each other, in case this key is updated, the car_id showing it opposite should show itself and the old one should be null. This was done with the following trigger.

```
CREATE TRIGGER IF NOT EXISTS update_car_id_after_update_appointment_id
AFTER
    UPDATE OF appointment_id ON car
BEGIN
   UPDATE appointment
   SET car_id=NEW.car_id
   WHERE appointment_id=NEW.appointment_id;

   UPDATE appointment
```

```
   SET car_id=null
   WHERE appointment_id=OLD.appointment_id AND OLD.car_id<>NEW.car_id
END;
```

- **Motor Table's Triggers**
  - **Validate Trigger**

    This trigger is written to check the ranges of the instance before adding a new instance to the motor table. The reason for using this instead of the check keyword is to clearly learn the error that occurs in any wrong situation.

```
CREATE TRIGGER IF NOT EXISTS validate_attribute_range_before_insert_motor
   BEFORE INSERT ON motor
BEGIN
   SELECT
      CASE
          WHEN NEW.engine_capacity<0 THEN RAISE (ABORT,'Invalid engine
capacity amount')
          WHEN NEW.engine_power<0 THEN RAISE (ABORT,'Invalid engine power
amount')
       END;
END;
```

- **Tyre Table's Triggers**
  - **Validate Trigger**

    This trigger is written to check the ranges of the instance before adding a new instance to the tyre table. The reason for using this instead of the check keyword is to clearly learn the error that occurs in any wrong situation.

```
CREATE TRIGGER IF NOT EXISTS validate_attribute_range_before_insert_tyre
   BEFORE INSERT ON tyre
BEGIN
   SELECT
      CASE
          WHEN (NEW.season not in ("Summer","Winter")) THEN RAISE
(ABORT,'Invalid season type')
       END;
END;
```

- **Equipment Table's Triggers**
  - **Validate Trigger**

    This trigger is written to check the ranges of the instance before adding a new instance to the equipment table. The reason for using this instead

of the check keyword is to clearly learn the error that occurs in any wrong situation.

```
CREATE TRIGGER IF NOT EXISTS
validate_attribute_range_before_insert_equipment
   BEFORE INSERT ON equipment
BEGIN
   SELECT
      CASE
          WHEN (NEW.equipment_level not in ("High","Medium","Low")) THEN
RAISE (ABORT,'Invalid equipment type')
       END;
END;
```

- **Truck Table's Triggers**
  - **Validate Trigger**

    This trigger is written to check the ranges of the instance before adding a new instance to the truck table. The reason for using this instead of the check keyword is to clearly learn the error that occurs in any wrong situation.

```
CREATE TRIGGER IF NOT EXISTS validate_attribute_range_before_insert_truck
   BEFORE INSERT ON truck
BEGIN
   SELECT
      CASE
          WHEN NEW.towing_capacity<0 THEN RAISE (ABORT,'Invalid towing
capacity amount')
          WHEN NEW.loading_capacity<0 THEN RAISE (ABORT,'Invalid loading
capacity amount')
       END;
END;
```

  - **Insert Trigger**

    Since the truck table is a car type, a car and truck pair should be formed, the foreign keys of which show each other due to the inheritance relationship during creation. Therefore, when truck is created, a trigger is written that creates a car. First, the car was created and a truck was connected with the car foreign key. Also, it was provided to connect the car in the newly created truck

```
CREATE TRIGGER IF NOT EXISTS add_car_after_add_truck
   AFTER INSERT ON truck
BEGIN
```

```
    INSERT INTO car (mark, weight, size, consumption, equipment_id,
appointment_id,tyre_id,motor_id,showroom_id,truck_id,sc_id,nc_id,suv_id,e
v_id)
    VALUES ("",0,0,0,0,null,0,0,0,NEW.truck_id,null,null,null,null);
    UPDATE truck SET

    car_id = CASE
         WHEN (SELECT seq FROM sqlite_sequence WHERE name="car")>0 THEN
(SELECT seq FROM sqlite_sequence WHERE name="car")+1
         ELSE 1
    END

    WHERE truck_id=NEW.truck_id;
END;
```

- **Delete Trigger**

  Since the truck is a type of car, when the test is deleted due to the inheritance relationship during the deletion, the row in the car table that connects to the test must be deleted. This was also done in the trigger below.

```
CREATE TRIGGER IF NOT EXISTS delete_car_after_delete_truck
   AFTER DELETE ON truck
BEGIN
    Delete from car where car_id=OLD.car_id;
END;
```

Sportscar Table's Triggers

- **Validate Trigger**

  This trigger is written to check the ranges of the instance before adding a new instance to the sportscar table. The reason for using this instead of the check keyword is to clearly learn the error that occurs in any wrong situation.

```
CREATE TRIGGER IF NOT EXISTS
validate_attribute_range_before_insert_sportscar
   BEFORE INSERT ON sportscar
BEGIN
   SELECT
      CASE
          WHEN NEW.max_speed<0 OR NEW.max_speed>400 THEN RAISE
(ABORT,'Invalid max speed amount')
       END;
END;
```

- **Insert Trigger**

Since the sportscar table is a car type, a car and sportscar pair should be formed, the foreign keys of which show each other due to the inheritance relationship during creation. Therefore, when sportscar is created, a trigger is written that creates a car. First, the car was created and a sportscar was connected with the car foreign key. Also, it was provided to connect the car in the newly created sportscar

```
CREATE TRIGGER IF NOT EXISTS add_car_after_add_sportscar
   AFTER INSERT ON sportscar
BEGIN
    INSERT INTO car (mark, weight, size, consumption, equipment_id,
appointment_id,tyre_id,motor_id,showroom_id,truck_id,sc_id,nc_id,suv_id,e
v_id)
    VALUES ("",0,0,0,0,null,0,0,0,null,NEW.sc_id,null,null,null);
    UPDATE sportscar SET

    car_id = CASE
         WHEN (SELECT seq FROM sqlite_sequence WHERE name="car")>0 THEN
(SELECT seq FROM sqlite_sequence WHERE name="car")+1
         ELSE 1
    END

    WHERE sc_id=NEW.sc_id;
END;
```

- **Delete Trigger**

  Since the sportscar is a type of car, when the test is deleted due to the inheritance relationship during the deletion, the row in the car table that connects to the test must be deleted. This was also done in the trigger below.

```
CREATE TRIGGER IF NOT EXISTS delete_car_after_delete_sportscar
   AFTER DELETE ON sportscar
BEGIN
    Delete from car where car_id=OLD.car_id;
END;
```

Normalcar Table's Triggers

- **Validate Trigger**

  This trigger is written to check the ranges of the instance before adding a new instance to the normalcar table. The reason for using this instead of the check keyword is to clearly learn the error that occurs in any wrong situation.

```
CREATE TRIGGER IF NOT EXISTS
validate_attribute_range_before_insert_normalcar
```

```
   BEFORE INSERT ON normalcar
BEGIN
   SELECT
      CASE
          WHEN NEW.person_capacity<0 THEN RAISE (ABORT,'Invalid person
capacity amount')
       END;
END;
```

- **Insert Trigger**

Since the normalcar table is a car type, a car and normalcar pair should be formed, the foreign keys of which show each other due to the inheritance relationship during creation. Therefore, when normalcar is created, a trigger is written that creates a car. First, the car was created and a normalcar was connected with the car foreign key. Also, it was provided to connect the car in the newly created normalcar

```
CREATE TRIGGER IF NOT EXISTS add_car_after_add_normalcar
   AFTER INSERT ON normalcar
BEGIN
    INSERT INTO car (mark, weight, size, consumption, equipment_id,
appointment_id,tyre_id,motor_id,showroom_id,truck_id,sc_id,nc_id,suv_id,e
v_id)
    VALUES ("",0,0,0,0,null,0,0,0,null,null,NEW.nc_id,null,null);
    UPDATE normalcar SET

    car_id = CASE
         WHEN (SELECT seq FROM sqlite_sequence WHERE name="car")>0 THEN
(SELECT seq FROM sqlite_sequence WHERE name="car")+1
         ELSE 1
    END

    WHERE nc_id=NEW.nc_id;
END;
```

- **Delete Trigger**

Since the normalcar is a type of car, when the test is deleted due to the inheritance relationship during the deletion, the row in the car table that connects to the test must be deleted. This was also done in the trigger below.

```
CREATE TRIGGER IF NOT EXISTS delete_car_after_delete_normalcar
   AFTER DELETE ON normalcar
BEGIN
    Delete from car where car_id=OLD.car_id;
```

```
END;
```

- ## Suv Table's Triggers
  - **Insert Trigger**

    Since the suv table is a car type, a car and suv pair should be formed, the foreign keys of which show each other due to the inheritance relationship during creation. Therefore, when suv is created, a trigger is written that creates a car. First, the car was created and a suv was connected with the car foreign key. Also, it was provided to connect the car in the newly created suv

```
CREATE TRIGGER IF NOT EXISTS add_car_after_add_suv
   AFTER INSERT ON suv
BEGIN
    INSERT INTO car (mark, weight, size, consumption, equipment_id,
appointment_id,tyre_id,motor_id,showroom_id,truck_id,sc_id,nc_id,suv_id,e
v_id)
    VALUES ("",0,0,0,0,null,0,0,0,null,null,null,NEW.suv_id,null);
    UPDATE suv SET

    car_id = CASE
         WHEN (SELECT seq FROM sqlite_sequence WHERE name="car")>0 THEN
(SELECT seq FROM sqlite_sequence WHERE name="car")+1
         ELSE 1
    END

    WHERE suv_id=NEW.suv_id;
END;
```

  - **Delete Trigger**

    Since the suv is a type of car, when the test is deleted due to the inheritance relationship during the deletion, the row in the car table that connects to the test must be deleted. This was also done in the trigger below.

```
CREATE TRIGGER IF NOT EXISTS delete_car_after_delete_suv
   AFTER DELETE ON suv
BEGIN
    Delete from car where car_id=OLD.car_id;
END;
```

- ## Electricvehicle Table's Triggers
  - **Validate Trigger**

    This trigger is written to check the ranges of the instance before adding a new instance to the electricvehicle table. The reason for using this

instead of the check keyword is to clearly learn the error that occurs in any wrong situation.

```
CREATE TRIGGER IF NOT EXISTS
validate_attribute_range_before_insert_electricvehicle
   BEFORE INSERT ON electricvehicle
BEGIN
   SELECT
      CASE
          WHEN NEW.battery_capacity<0 THEN RAISE (ABORT,'Invalid battery
capacity amount')
       END;
END;
```

- **Insert Trigger**

Since the electricvehicle table is a car type, a car and electricvehicle pair should be formed, the foreign keys of which show each other due to the inheritance relationship during creation. Therefore, when electricvehicle is created, a trigger is written that creates a car. First, the car was created and a electricvehicle was connected with the car foreign key. Also, it was provided to connect the car in the newly created electricvehicle

```
CREATE TRIGGER IF NOT EXISTS add_car_after_add_ev
   AFTER INSERT ON electricvehicle
BEGIN
    INSERT INTO car (mark, weight, size, consumption, equipment_id,
appointment_id,tyre_id,motor_id,showroom_id,truck_id,sc_id,nc_id,suv_id,e
v_id)
    VALUES ("",0,0,0,0,null,0,0,0,null,null,null,null,NEW.ev_id);
    UPDATE electricvehicle SET

    car_id = CASE
         WHEN (SELECT seq FROM sqlite_sequence WHERE name="car")>0 THEN
(SELECT seq FROM sqlite_sequence WHERE name="car")+1
         ELSE 1
    END

    WHERE ev_id=NEW.ev_id;
END;
```

- **Delete Trigger**

Since the electricvehicle is a type of car, when the test is deleted due to the inheritance relationship during the deletion, the row in the car table that connects to the test must be deleted. This was also done in the trigger below.

```
CREATE TRIGGER IF NOT EXISTS delete_car_after_delete_ev
   AFTER DELETE ON electricvehicle
BEGIN
    Delete from car where car_id=OLD.car_id;
END;
```

- **Showroom Table's Triggers**
  - **Validate Trigger**

    This trigger is written to check the ranges of the instance before adding a new instance to the showroom table. The reason for using this instead of the check keyword is to clearly learn the error that occurs in any wrong situation.

```
CREATE TRIGGER IF NOT EXISTS
validate_attribute_range_before_insert_showroom
   BEFORE INSERT ON showroom
BEGIN
   SELECT
      CASE
          WHEN NEW.capacity<0 THEN RAISE (ABORT,'Invalid showroom
capacity amount')
       END;
END;
```

- **Department Table's Triggers**
  - **Validate Trigger**

    This trigger is written to check the ranges of the instance before adding a new instance to the department table. The reason for using this instead of the check keyword is to clearly learn the error that occurs in any wrong situation.

```
CREATE TRIGGER IF NOT EXISTS
validate_attribute_range_before_insert_department
   BEFORE INSERT ON department
BEGIN
   SELECT
      CASE
          WHEN NEW.budget<0 THEN RAISE (ABORT,'Invalid budget amount')
       END;
END;
```

- **Employee Table's Triggers**
  - **Validate Trigger**

This trigger is written to check the ranges of the instance before adding a new instance to the employee table. The reason for using this instead of the check keyword is to clearly learn the error that occurs in any wrong situation.

```
CREATE TRIGGER IF NOT EXISTS
validate_attribute_range_before_insert_employee
   BEFORE INSERT ON employee
BEGIN
   SELECT
      CASE
          WHEN NEW.salary<0 THEN RAISE (ABORT,'Invalid salary amount')
          WHEN (NEW.gender not in ("Female","Male")) THEN RAISE
(ABORT,'Invalid gender type')
       END;
END;
```

- **Technician Table's Triggers**
  - **Insert Trigger**

    Since the technician table is a employee type, a employee and technician pair should be formed, the foreign keys of which show each other due to the inheritance relationship during creation. Therefore, when technician is created, a trigger is written that creates a employee. First, the employee was created and a technician was connected with the employee foreign key. Also, it was provided to connect the car in the newly created technician

```
CREATE TRIGGER IF NOT EXISTS add_employee_after_add_technician
   AFTER INSERT ON technician
BEGIN
    INSERT INTO employee (name, address, salary, gender, tech_id,
mng_id,sr_id,rec_id,lr_id,dept_name)
    VALUES ("","",0,"Female",NEW.tech_id,null,null,null,null,"");
    UPDATE technician SET

    employee_id = CASE
         WHEN (SELECT seq FROM sqlite_sequence WHERE name="employee")>0
THEN (SELECT seq FROM sqlite_sequence WHERE name="employee")+1
         ELSE 1
    END

    WHERE tech_id=NEW.tech_id;
END;
```

  - **Delete Trigger**

Since the technician is a type of employee, when the test is deleted due to the inheritance relationship during the deletion, the row in the employee table that connects to the test must be deleted. This was also done in the trigger below.

```
CREATE TRIGGER IF NOT EXISTS delete_employee_after_delete_technician
   AFTER DELETE ON technician
BEGIN
    Delete from employee where employee_id=OLD.employee_id;
END;
```

- **Manager Table's Triggers**
  - **Insert Trigger**

    Since the manager table is a employee type, a employee and manager pair should be formed, the foreign keys of which show each other due to the inheritance relationship during creation. Therefore, when manager is created, a trigger is written that creates a employee. First, the employee was created and a manager was connected with the employee foreign key. Also, it was provided to connect the car in the newly created manager

```
CREATE TRIGGER IF NOT EXISTS add_employee_after_add_manager
   AFTER INSERT ON manager
BEGIN
    INSERT INTO employee (name, address, salary, gender, tech_id,
mng_id,sr_id,rec_id,lr_id,dept_name)
    VALUES ("","",0,"Female",null,NEW.mng_id,null,null,null,"");
    UPDATE manager SET

    employee_id = CASE
         WHEN (SELECT seq FROM sqlite_sequence WHERE name="employee")>0
THEN (SELECT seq FROM sqlite_sequence WHERE name="employee")+1
         ELSE 1
    END

    WHERE mng_id=NEW.mng_id;
END;
```

  - **Delete Trigger**

    Since the manager is a type of employee, when the test is deleted due to the inheritance relationship during the deletion, the row in the employee table that connects to the test must be deleted. This was also done in the trigger below.

```
CREATE TRIGGER IF NOT EXISTS delete_employee_after_delete_manager
   AFTER DELETE ON manager
BEGIN
```

```
    Delete from employee where employee_id=OLD.employee_id;
END;
```

- ## Sales Responsible Table's Triggers
  - **Insert Trigger**

    Since the sales responsible table is a employee type, a employee and sales responsible pair should be formed, the foreign keys of which show each other due to the inheritance relationship during creation. Therefore, when sales responsible is created, a trigger is written that creates a employee. First, the employee was created and a sales responsible was connected with the employee foreign key. Also, it was provided to connect the car in the newly created sales responsible

```
CREATE TRIGGER IF NOT EXISTS add_employee_after_add_sales_responsible
   AFTER INSERT ON sales_responsible
BEGIN
    INSERT INTO employee (name, address, salary, gender, tech_id,
mng_id,sr_id,rec_id,lr_id,dept_name)
    VALUES ("","",0,"Female",null,null,NEW.sr_id,null,null,"");
    UPDATE sales_responsible SET

    employee_id = CASE
         WHEN (SELECT seq FROM sqlite_sequence WHERE name="employee")>0
THEN (SELECT seq FROM sqlite_sequence WHERE name="employee")+1
         ELSE 1
    END

    WHERE sr_id=NEW.sr_id;
END;
```

  - **Delete Trigger**

    Since the sales responsible is a type of employee, when the test is deleted due to the inheritance relationship during the deletion, the row in the employee table that connects to the test must be deleted. This was also done in the trigger below.

```
CREATE TRIGGER IF NOT EXISTS
delete_employee_after_delete_sales_responsible
   AFTER DELETE ON sales_responsible
BEGIN
    Delete from employee where employee_id=OLD.employee_id;
END;
```

- ## Reception Table's Triggers
  - **Insert Trigger**

Since the reception table is a employee type, a employee and reception pair should be formed, the foreign keys of which show each other due to the inheritance relationship during creation. Therefore, when reception is created, a trigger is written that creates a employee. First, the employee was created and a reception was connected with the employee foreign key. Also, it was provided to connect the car in the newly created reception

```
CREATE TRIGGER IF NOT EXISTS add_employee_after_add_reception
   AFTER INSERT ON reception
BEGIN
    INSERT INTO employee (name, address, salary, gender, tech_id,
mng_id,sr_id,rec_id,lr_id,dept_name)
    VALUES ("","",0,"Female",null,null,null,NEW.rec_id,null,"");
    UPDATE reception SET

    employee_id = CASE
         WHEN (SELECT seq FROM sqlite_sequence WHERE name="employee")>0
THEN (SELECT seq FROM sqlite_sequence WHERE name="employee")+1
         ELSE 1
    END

    WHERE rec_id=NEW.rec_id;
END;
```

- **Delete Trigger**

  Since the reception is a type of employee, when the test is deleted due to the inheritance relationship during the deletion, the row in the employee table that connects to the test must be deleted. This was also done in the trigger below.

```
CREATE TRIGGER IF NOT EXISTS delete_employee_after_delete_reception
   AFTER DELETE ON reception
BEGIN
    Delete from employee where employee_id=OLD.employee_id;
END;
```

Leasing Responsible Table's Triggers

- **Insert Trigger**

  Since the leasing responsible table is a employee type, a employee and leasing responsible pair should be formed, the foreign keys of which show each other due to the inheritance relationship during creation. Therefore, when leasing responsible is created, a trigger is written that creates a employee. First, the employee was created and a leasing responsible was connected with the employee foreign key. Also, it was provided to connect the car in the newly created leasing responsible

```
CREATE TRIGGER IF NOT EXISTS add_employee_after_add_leasing_responsible
   AFTER INSERT ON leasing_responsible
BEGIN
    INSERT INTO employee (name, address, salary, gender, tech_id,
mng_id,sr_id,rec_id,lr_id,dept_name)
    VALUES ("","",0,"Female",null,null,null,null,NEW.lr_id,"");
    UPDATE leasing_responsible SET

    employee_id = CASE
         WHEN (SELECT seq FROM sqlite_sequence WHERE name="employee")>0
THEN (SELECT seq FROM sqlite_sequence WHERE name="employee")+1
         ELSE 1
    END

    WHERE lr_id=NEW.lr_id;
END;
```

- **Delete Trigger**

  Since the leasing responsible is a type of employee, when the test is deleted due to the inheritance relationship during the deletion, the row in the employee table that connects to the test must be deleted. This was also done in the trigger below.

```
CREATE TRIGGER IF NOT EXISTS
delete_employee_after_delete_leasing_responsible
   AFTER DELETE ON leasing_responsible
BEGIN
    Delete from employee where employee_id=OLD.employee_id;
END;
```

• VIEWS

Views were made to combine information in some tables in a meaningful way and provide quick access. In general, the views that emerged as a result of combining the inherited child tables with the parent tables were made. Apart from this, views that reveal all the features of a car were made. Similarly, views revealing Appointment and its invoice were written. In total, 20 different views were written.

○ Customer Table's Views

As mentioned above, views that combine customer types and the main table were created in the customer's views. Thus, all customer information for each customer has become instantaneous.

```
CREATE VIEW IF NOT EXISTS [Leasing Customer View] AS
    SELECT
        customer.customer_id,name,money,gender,email,customer.lc_id
    FROM
        customer INNER JOIN leasing_customer ON leasing_customer.customer_id
= customer.customer_id;


CREATE VIEW IF NOT EXISTS [Saling Customer View] AS
    SELECT
        customer.customer_id,name,money,gender,email,customer.sc_id
    FROM
        customer INNER JOIN saling_customer ON saling_customer.customer_id =
customer.customer_id;

CREATE VIEW IF NOT EXISTS [Maintanence Customer View] AS
    SELECT
        customer.customer_id,name,money,gender,email,customer.mc_id
    FROM
        customer INNER JOIN maintanence_customer ON
maintanence_customer.customer_id = customer.customer_id;

CREATE VIEW IF NOT EXISTS [Test Customer View] AS
    SELECT
        customer.customer_id,name,money,gender,email,customer.tc_id
    FROM
        customer INNER JOIN test_customer ON test_customer.customer_id =
customer.customer_id;
```

- **Sample Running Of Customer – Leasing Customer**

custo…	name	money	gender	email	tc_id	mc_id	lc_id	sc_id
1	Fatih Selim Yakar	10000	Male	fatihselim…			1	
2	Hülya Nur YAKAR	8996	Female	deneme…			2	
3	Coşkun YAKAR	499997	Male	none@no…				1
4	Ayşe YAKAR	123238	Female	none@no…	1			

Figure 1 – Customer Table

lc_id	customer_id
1	1
2	2

Figure 2 – Leasing Customer Table

customer_id	name	money	gender	email	lc_id
1	Fatih Selim Yakar	10000	Male	fatihselimyakar@…	1
2	Hülya Nur YAKAR	8996	Female	deneme@denem…	2

Figure 3 – Leasing Customer View

- **Appointment Table's Views**

Views that combine appointment types, cars and the main table were created in the appointment's views. Thus, all appointment information for each appointment has become instantaneous.

```
CREATE VIEW IF NOT EXISTS [Maintanence Appointment View] AS
    SELECT

appointment.appointment_id,appointment.date_detail,maintanence.m_id,maintane
nce.delivery_time,car.car_id,car.mark
    FROM
        (maintanence INNER JOIN appointment INNER JOIN car
        ON maintanence.m_id = appointment.m_id  AND appointment.car_id =
car.car_id);


CREATE VIEW IF NOT EXISTS [Leasing Appointment View] AS
    SELECT

appointment.appointment_id,appointment.date_detail,leasing.leasing_id,leasin
g.delivery_date,car.car_id,car.mark
    FROM
        (leasing INNER JOIN appointment INNER JOIN car
        ON leasing.leasing_id = appointment.leasing_id  AND
appointment.car_id = car.car_id);

CREATE VIEW IF NOT EXISTS [Sale Appointment View] AS
    SELECT

appointment.appointment_id,appointment.date_detail,sale.sale_id,car.car_id,c
ar.mark
    FROM
        (sale INNER JOIN appointment INNER JOIN car
        ON sale.sale_id = appointment.sale_id  AND appointment.car_id =
car.car_id);

CREATE VIEW IF NOT EXISTS [Test Appointment View] AS
    SELECT

appointment.appointment_id,appointment.date_detail,test.test_id,test.test_in
terval,car.car_id,car.mark
    FROM
        (test INNER JOIN appointment INNER JOIN car
```

```
        ON test.test_id = appointment.test_id  AND appointment.car_id =
car.car_id);
```

- **Sample Running Of Appointment – Maintanence Appointment**

app...	date_detail	bill_id	custo...	m_id	sale_id	leasing_id	test_id	rec_id	car_id
1	2021-6-6 12:14:...	1	0	1					3
2	2021-10-6 14:00...	2	0	2					1
3	2021-9-9 00:00:...	3	0				1		2

Figure 4 – Appointment Table

m_id	delivery_time	tech_id	appointment_id
1	120	1	1
2	560	2	2

Figure 5 – Maintanence Table

c..	mark	we...	size	co...	equi...	appoint...	tyre...	motor...	sho...	truck...	sc...	nc...	su
1	Volksw...	1350	4850	7	1	2	1	1	1	1			
2	Fiat - 5...	1000	3500	5	1	3	1	1	1		1		
3	Ford - ...	2000	5150	15	1	1	1	1	1				1

Figure 6 – Car Table

appointment_id	date_detail	m_id	delivery_time	car_id	mark
1	2021-6-6 12:14:00.000	1	120	3	Ford - Ranger
2	2021-10-6 14:00:00.000	2	560	1	Volkswagen - Pas...

Figure 7 – Maintanence Appointment View

- **Car Table's Views**

Views that combine inherited car types, motor, tyre, equipment and the main table were created in the car's views. Thus, all car information for each appointment has become instantaneous.

```
CREATE VIEW IF NOT EXISTS [Truck Car View] AS
    SELECT

car.car_id,car.mark,car.weight,car.size,car.consumption,truck.towing_capacit
y,truck.loading_capacity,motor.engine_capacity,motor.engine_power,tyre.tyre_
mark,tyre.season,equipment.equipment_level,equipment.equipment_options
    FROM
        car
        INNER JOIN truck ON car.car_id = truck.car_id
        INNER JOIN motor ON motor.motor_id = car.motor_id
```

```
        INNER JOIN tyre ON tyre.tyre_id = car.tyre_id
        INNER JOIN equipment ON equipment.equipment_id = car.equipment_id;


CREATE VIEW IF NOT EXISTS [Sports Car View] AS
    SELECT

car.car_id,car.mark,car.weight,car.size,car.consumption,sportscar.max_speed,
motor.engine_capacity,motor.engine_power,tyre.tyre_mark,tyre.season,equipmen
t.equipment_level,equipment.equipment_options
    FROM
        car
        INNER JOIN sportscar ON car.car_id = sportscar.car_id
        INNER JOIN motor ON motor.motor_id = car.motor_id
        INNER JOIN tyre ON tyre.tyre_id = car.tyre_id
        INNER JOIN equipment ON equipment.equipment_id = car.equipment_id;

CREATE VIEW IF NOT EXISTS [Normal Car View] AS
    SELECT

car.car_id,car.mark,car.weight,car.size,car.consumption,normalcar.person_cap
acity,motor.engine_capacity,motor.engine_power,tyre.tyre_mark,tyre.season,eq
uipment.equipment_level,equipment.equipment_options
    FROM
        car
        INNER JOIN normalcar ON car.car_id = normalcar.car_id
        INNER JOIN motor ON motor.motor_id = car.motor_id
        INNER JOIN tyre ON tyre.tyre_id = car.tyre_id
        INNER JOIN equipment ON equipment.equipment_id = car.equipment_id;

CREATE VIEW IF NOT EXISTS [Suv Car View] AS
    SELECT

car.car_id,car.mark,car.weight,car.size,car.consumption,suv.is_awd,motor.eng
ine_capacity,motor.engine_power,tyre.tyre_mark,tyre.season,equipment.equipme
nt_level,equipment.equipment_options
    FROM
        car
        INNER JOIN suv ON car.car_id = suv.car_id
        INNER JOIN motor ON motor.motor_id = car.motor_id
        INNER JOIN tyre ON tyre.tyre_id = car.tyre_id
        INNER JOIN equipment ON equipment.equipment_id = car.equipment_id;


CREATE VIEW IF NOT EXISTS [Electric Car View] AS
    SELECT

car.car_id,car.mark,car.weight,car.size,car.consumption,electricvehicle.batt
```

```
ery_capacity,motor.engine_capacity,motor.engine_power,tyre.tyre_mark,tyre.se
ason,equipment.equipment_level,equipment.equipment_options
    FROM
        car
        INNER JOIN electricvehicle ON car.car_id = electricvehicle.car_id
        INNER JOIN motor ON motor.motor_id = car.motor_id
        INNER JOIN tyre ON tyre.tyre_id = car.tyre_id
        INNER JOIN equipment ON equipment.equipment_id = car.equipment_id;
```

- **Sample Running Of Car – Truck Car**

c..	mark	we...	size	co...	equi...	appoint...	tyre...	motor...	sho...	truck...	sc...	nc...	su
1	Volksw...	1350	4850	7	2	2	2	2	1	1			
2	Fiat - 5...	1000	3500	5	1	3	2	1	1		1		
3	Ford - ...	2000	5150	15	1	1	1	1	1				1

Figure 8 – Car Table

truck_id	towing_capacity	loading_capacity	car_id
1	500	500	1

Figure 9 – Truck Table

equipment_id	equipment_level	equipment_options
1	High	Red Matte Color, Shinny Rims
2	Low	Automatic Gearbox

Figure 10 – Equipment Table

tyre_id	tyre_mark	season
1	Michelin	Summer
2	Bridgestone	Winter

Figure 11 – Tyre Table

motor_id	engine_capacity	engine_power
1	2.5	320
2	4.4	550

Figure 12 – Motor Table

c...	mark	wei...	size	con...	towi...	load...	engi...	engi...	tyre...	sea...	equi...	equipmen...
1	Volks...	1350	4850	7	500	500	4.4	550	Bridg...	Winter	Low	Automatic ...

Figure 13 – Truck Car View

- **Employee Table's Views**

Views that combine inherited emplotee types and the main table were created in the employee's views. Thus, all employee information for each appointment has become instantaneous.

```
CREATE VIEW IF NOT EXISTS [Technician Employee View] AS
    SELECT

employee.employee_id,employee.name,employee.address,employee.salary,employee
.gender,technician.mastery,employee.dept_name
    FROM
        employee
        INNER JOIN technician ON employee.tech_id = technician.tech_id;

CREATE VIEW IF NOT EXISTS [Manager Employee View] AS
    SELECT

employee.employee_id,employee.name,employee.address,employee.salary,employee
.gender,manager.graduated_unv,manager.h_name,employee.dept_name
    FROM
        employee
        INNER JOIN manager ON employee.mng_id = manager.mng_id;

CREATE VIEW IF NOT EXISTS [Sales Responsible Employee View] AS
    SELECT

employee.employee_id,employee.name,employee.address,employee.salary,employee
.gender,sales_responsible.car_type,employee.dept_name
    FROM
        employee
        INNER JOIN sales_responsible ON employee.sr_id =
sales_responsible.sr_id;

CREATE VIEW IF NOT EXISTS [Reception Employee View] AS
    SELECT

employee.employee_id,employee.name,employee.address,employee.salary,employee
.gender,reception.phone_number,employee.dept_name
    FROM
        employee
        INNER JOIN reception ON employee.rec_id = reception.rec_id;

CREATE VIEW IF NOT EXISTS [Leasing Responsible Employee View] AS
    SELECT

employee.employee_id,employee.name,employee.address,employee.salary,employee
.gender,leasing_responsible.lr_id,employee.dept_name
    FROM
```

```
        employee
        INNER JOIN leasing_responsible ON employee.lr_id =
leasing_responsible.lr_id;
```

- **Sample Running Of Employee – Manager**

mng_id	graduated_unv	employee_id	h_name
1	Gebze Technical University	1	Volkswagen Automobile Group

Figure 14 - Manager Table

em...	name	address	salary	gender	tech_id	mng_id	sr_id	rec_id	lr_id	dept_na...
1	Yönetici H...	Istanbu...	10000	Female		1				Administr...

Figure 15 - Employee Table

employ...	name	address	salary	gender	graduated...	h_name	dept_name
1	Yönetici Ha...	Istanbul Tek...	10000	Female	Gebze Tech...	Volkswagen...	Administration D...

Figure 16 – Manager Employee View

○ Special Views

The views in this section were made for the important main topics, Car and Appointment. Several inert tables and other tables were combined to get all the information in these sections. The most important information for the tables in this main topic has been listed.

```
--SPECIAL VIEWS
CREATE VIEW IF NOT EXISTS [Appointments with Customer and Bill View] AS
    SELECT

customer.customer_id,name,money,gender,email,appointment.appointment_id,appo
intment.date_detail,bill.bill_date,bill.cost
    FROM
        (customer INNER JOIN appointment INNER JOIN bill
        ON appointment.customer_id = customer.customer_id  AND
appointment.appointment_id = bill.appointment_id);


CREATE VIEW IF NOT EXISTS [Cars with Motor,Tyre and Equipment View] AS
    SELECT

car.car_id,mark,weight,size,consumption,motor.motor_id,motor.engine_capacity
,motor.engine_power,tyre.tyre_id,tyre.tyre_mark,tyre.season,equipment.equipm
ent_id,equipment.equipment_level,equipment.equipment_options
    FROM
        ((car INNER JOIN motor ON car.motor_id = motor.motor_id)
```

```
        INNER JOIN tyre ON car.tyre_id = tyre.tyre_id)
        INNER JOIN equipment ON car.equipment_id = equipment.equipment_id;
```

- **Sample Running Of Appointments Special View**

custo…	name	money	gender	email	tc_id	mc_id	lc_id	sc_id
1	Fatih Selim Yakar	10000	Male	fatihselim…			1	
2	Hülya Nur YAKAR	8996	Female	deneme…			2	
3	Coşkun YAKAR	499997	Male	none@no…				1
4	Ayşe YAKAR	123238	Female	none@no…	1			

Figure 17 – Customer Table

app…	date_detail	bill_id	customer_id	m_id	sale_id	leasing_id	test_id	rec_id
1	2021-6-6 12:14:00.000	1	1	1				
2	2021-10-6 14:00:00.000	2	2	2				
3	2021-9-9 00:00:00.000	3	3				1	

Figure 18 – Appointment Table

bill_id	bill_date	cost	appointment_id	customer_id
1	2025-12-12 00:00:00.000	500000	1	1
2	2026-12-12 00:00:00.000	6321445	2	2
3	2028-12-12 00:00:00.000	1250	3	3

Figure 19 – Bill Table

custo…	name	money	gender	email	appoint…	date_detail	bill_date	cost
1	Fatih Seli…	10000	Male	fatihselim…	1	2021-6-6 …	2025-12-1…	500000
2	Hülya Nur…	8996	Female	deneme…	2	2021-10-6…	2026-12-1…	6321445
3	Coşkun Y…	499997	Male	none@no…	3	2021-9-9 …	2028-12-1…	1250

Figure 20 – Appointments with Customer and Bill View

- **Sample Running Of Cars Special View**

c..	mark	we…	size	co…	equi…	appoint…	tyre…	motor…	sho…	truck…	sc…	nc…	su
1	Volksw…	1350	4850	7	2	2	2	2	1	1			
2	Fiat - 5…	1000	3500	5	1	3	2	1	1		1		
3	Ford - …	2000	5150	15	1	1	1	1	1				1

Figure 21 – Cars Table

equipment_id	equipment_level	equipment_options
1	High	Red Matte Color, Shinny Rims
2	Low	Automatic Gearbox

Figure 22 – Equipment Table

tyre_id	tyre_mark	season
1	Michelin	Summer
2	Bridgestone	Winter

Figure 23 – Tyre Table

motor_id	engine_capacity	engine_power
1	2.5	320
2	4.4	550

Figure 24 – Motor Table

car_id	mark	wei…	size	co…	mo…	en…	en…	tyr…	tyr…	sea…	eq…	eq…	equip…
1	Volk…	1350	4850	7	2	4.4	550	2	Brid…	Winter	2	Low	Autom…
2	Fiat …	1000	3500	5	1	2.5	320	2	Brid…	Winter	1	High	Red M…
3	Ford…	2000	5150	15	1	2.5	320	1	Mich…	Sum…	1	High	Red M…

Figure 25 – Cars with Motor,Tyre and Equipment View

- # HIGH LEVEL PROGRAMMING CONNECTION WITH DATABASE

In this part, a connection was made with the database using the Java programming language. This connection was provided with JDBC, which we saw in the lesson. JDBC stands for **J**ava **D**ata**b**ase **C**onnectivity, which is a standard Java API for database-independent connectivity between the Java programming language and a wide range of databases.

The JDBC library includes APIs for each of the tasks mentioned below that are commonly associated with database usage.

- Making a connection to a database.
- Creating SQL or MySQL statements.
- Executing SQL or MySQL queries in the database.
- Viewing & Modifying the resulting records.

  - ## JDBC Connection

    In this part, I wrote a java class called CarGalleryConnection to connect with the database with the help of jdbc. In this written class, the url of the database and the jdbc driver were specified, and a connection was made with the database.

```
package com.project.cargallery.db;
```

```
//Step 1: Use interfaces from java.sql package
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.SQLException;

public class CarGalleryConnection {
  private static CarGalleryConnection instance = new CarGalleryConnection();
  public static final String URL =
"jdbc:sqlite:/Users/fatihselimyakar/Desktop/cargallery_filled.db";
  public static final String DRIVER_CLASS = "org.sqlite.JDBC";

  private CarGalleryConnection() {
      try {
          Class.forName(DRIVER_CLASS);
      } catch (ClassNotFoundException e) {
          e.printStackTrace();
          System.out.println(e);
      }
  }

  private Connection createConnection() {

      Connection connection = null;
      try {
          //Step 3: Establish Java MySQL connection
          connection = DriverManager.getConnection(URL);
      } catch (SQLException e) {
          System.out.println("ERROR: Unable to Connect to Database.");
          System.out.println(e);
      }
      return connection;
  }

  public static Connection getConnection() {
      return instance.createConnection();
  }
}
```

- **Model Interface**

An interface was written in order to "implements" the models written for each table that will be specified later. In this interface, there were function prototypes written to return the queries to be executed as strings.

```
package com.project.cargallery.interfaces;

import java.sql.ResultSet;
import java.sql.SQLException;

public interface ModelInterface<T> {
   public T exportTableInstance(ResultSet rs) throws SQLException;
   public String getSelectQuery(int id);
   public String getSelectAllQuery();
   public String getDeleteQuery(int id);
   public String getDeleteAllQuery();
   public String getInsertQuery(T object);
   public String getUpdateQuery(T object,int id);
}
```

- **DB Manipulating Methods**

In this part, the methods that the Database will run and the implementation of those methods were included. These methods included the following functions: Select, select all, delete, delete all, update, add, select for views, manual querying. In addition, the part that entered this class as a private object was forced to derive from ModelInterface, from which models derive.

```
package com.project.cargallery.db;

import java.sql.Connection;
import java.sql.ResultSet;
import java.sql.ResultSetMetaData;
import java.sql.SQLException;
import java.sql.Statement;
import java.util.ArrayList;
import java.util.List;

import com.project.cargallery.interfaces.ModelInterface;

public class CarGalleryDbMethods<T extends ModelInterface<T>> {
    private T privateObject;

    public CarGalleryDbMethods(T object){..}

    public T getInstance(int id)  {..}

    public void deleteInstance(int id)  {..}

    public List<T> getAllIntances()  {..}

    public void addInstance(T object)  {..}

    public void deleteAlLInstances()  {..}

    public void updateInstance(T object,int id)  {..}

    public String getAllIntancesForViews()  {..}

    public String getAllIntancesManually(String queryString)  {..}

    public void updateManually(String queryString)  {..}

}
```

Since the implementation of each method will take up a lot of space, these parts are indicated with {..} above. However, here are two examples that use the executeUpdate and executeQuery functions and are very similar to other implementations:

```
public T getInstance(int id)  {
        ResultSet rs = null;
        Connection connection = null;
        Statement statement = null;

        String query = privateObject.getSelectQuery(id);
        try {
```

```
            connection = CarGalleryConnection.getConnection();
            statement = connection.createStatement();
            rs = statement.executeQuery(query);

            if (rs.next()) {
                privateObject = privateObject.exportTableInstance(rs);
            }
        } catch (SQLException e) {
            e.printStackTrace();
        } finally {
            if (connection != null) {
                try {
                    connection.close();
                } catch (SQLException e) {
                    e.printStackTrace();
                }
            }
        }
        return privateObject;
    }
```

```
public void deleteInstance(int id)  {
        Connection connection = null;
        Statement statement = null;

        String query = privateObject.getDeleteQuery(id);
        try {
            connection = CarGalleryConnection.getConnection();
            statement = connection.createStatement();
            statement.executeUpdate(query);

        } catch (SQLException e) {
            e.printStackTrace();
        } finally {
            if (connection != null) {
                try {
                    connection.close();
                } catch (SQLException e) {
                    e.printStackTrace();
                }
            }
        }
    }
```

- ## Table Models

In this part, models are written that can both return the queries that will work in the methods and hold the outputs obtained as a result of any operation so that the above db methods can work. In this section, 29 classes were written separately for each table and 1 class was written for views. These models were implementing the ModelInterface model written above. Below is the Java class model of Bill table as an example.

```
package com.project.cargallery.models;

import java.sql.Date;
import java.sql.ResultSet;
import java.sql.SQLException;
```

```
import com.project.cargallery.interfaces.ModelInterface;

public class Bill implements ModelInterface<Bill>{
    private int billId;
    private Date billDate;
    private int cost;
    private int appointmentId;
    private int customerId;

    public Bill() {}

    public Bill(int billId, Date billDate, int cost, int appointmentId, int
customerId) {..}

    public int getBillId() {..}
    public void setBillId(int billId) {..}
    public Date getBillDate() {..}
    public void setBillDate(Date billDate) {..}
    public int getCost() {..}
    public void setCost(int cost) {..}
    public int getAppointmentId() {..}
    public void setAppointmentId(int appointmentId) {..}
    public int getCustomerId() {..}
    public void setCustomerId(int customerId) {..}
    @Override
    public String toString() {..}
    @Override
    public Bill exportTableInstance(ResultSet rs) throws SQLException {
            return new Bill(rs.getInt("bill_id"),rs.getDate("bill_date"),
rs.getInt("cost"), rs.getInt("appointment_id"),rs.getInt("customer_id"));
    }
    @Override
    public String getSelectQuery(int id) {
            return "SELECT * FROM bill WHERE bill_id=" + id;
    }
    @Override
    public String getSelectAllQuery() {
            return "SELECT * FROM bill";
    }
    @Override
    public String getDeleteQuery(int id) {
            return "DELETE FROM bill WHERE bill_id=" + id;
    }
    @Override
    public String getDeleteAllQuery() {
            return "DELETE FROM sqlite_sequence WHERE name='bill';";
    }
    @Override
    public String getInsertQuery(Bill object) {
            return "INSERT INTO bill (bill_date, cost, appointment_id,
customer_id) VALUES
("+"'"+object.getBillDate()+"'"+","+object.getCost()+","+object.getAppointme
ntId()+","+object.getCustomerId()+")";
    }

    @Override
    public String getUpdateQuery(Bill object,int id) {
            return "UPDATE bill SET
bill_id='"+object.getBillId()+"',bill_date='"+object.getBillId()+"',
```

```
cost='"+object.getCost()+"', appointment_id='"+object.getAppointmentId()+"',
customer_id='"+object.getCustomerId()+"' WHERE bill_id="+id;
    }


}
```

- ## General Structure Of Java Project

Figure 26 – Project Structure

- **Sample Running in Eclipse**

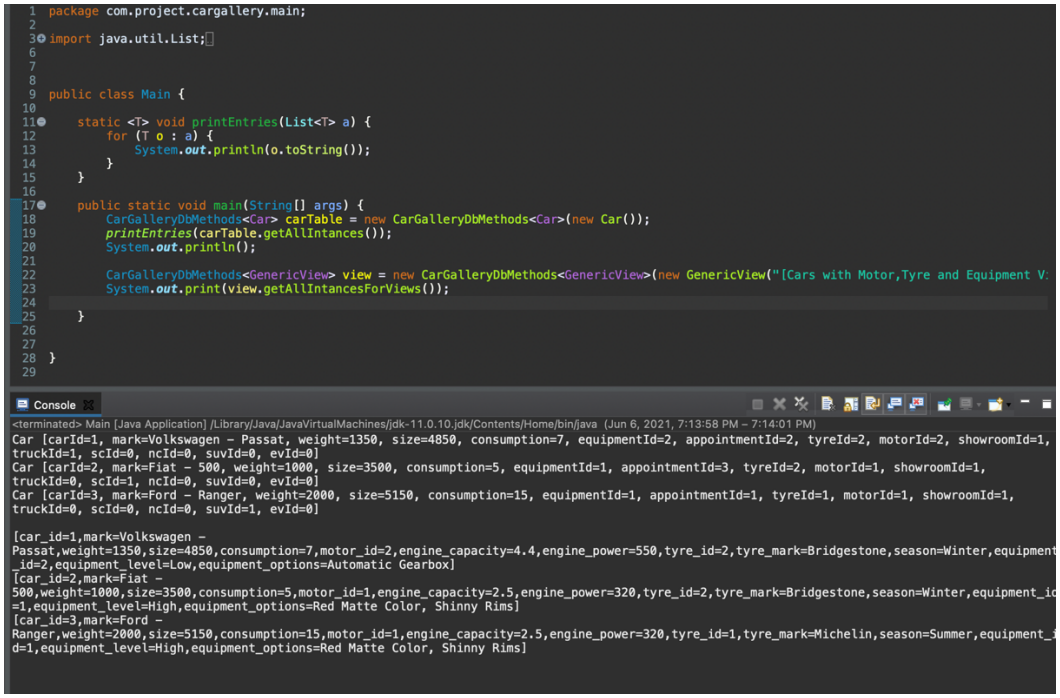


Figure 27 – Sample Running

• USING PROGRAMS AND TOOLS

- Sqlite JDBC Version 3.30.1 for JDBC Connection
- Java SE 11 (11.0.10) for Java Connection Project
- Eclipse IDE Version 4.19.0 for Java Connection Project
- DBeaver Version 21.0.4 for Drawing ER diagram
- SQLiteStudio Version 3.3.3 for Writing SQL queries
- SQLite Version 3 for DBMS system
- Some Eclipse plug-in for GUI

• REFERENCES

- Stackoverflow [Online], https://stackoverflow.com/questions/* [Access Date: 2021].

- Geeksforgeeks [Online], https://www.geeksforgeeks.org/* [Access Date: 2021].
- SQLiteOnlie [Online], https://sqliteonline.com/ [Access Date: 2021].
- W3 Schools [Online], https://www.w3schools.com/sql/ [Access Date: 2021].
- Sqlite [Online], https://www.sqlite.org/about.html [Access Date: 2021].
- Tutorialspoint [Online], https://www.tutorialspoint.com/jdbc/jdbc-introduction.htm [Access Date: 2021].
- Database System Concepts Seventh Edition, *Avi Silberschatz, Henry F. Korth, S. Sudarshan*
- Sqlitetutorial [Online], https://www.sqlitetutorial.net/ [Access Date: 2021].

- **NOTES**
  - Tables, which are the only primary keys in the DBMS system, are made to provide a comprehensive design considering the future of the system. In other words, it was made so that new attributes can be easily added when necessary.
  - The infrastructure has been built for the interface expectations specified in the project requirements. You can easily see this in the DB Methods section in the Java project. (“Create the user interface of at least 5 modules using any programming language. Choose the interfaces according to the questions below. Create 1 outer right, 1 outer left, full outer querry on user interface”)
  - "Add additional details about the database if there is any." and "Use inheritance for the tables." parts of it have been made, specified as "additional" in the project requirements
  - All the mentioned implementations have been tried.
  - As a result of the research, it was learned that the primary and foreign keys are indexed by default in sqlite, so different attributes in the tables were created as indexes.