

ITC607 Data Management

Assignment 2



Faculty of Health, Humanities and Computing

Name / Student ID: Shawn Chen / 2017003399

Date: 16 June 2019

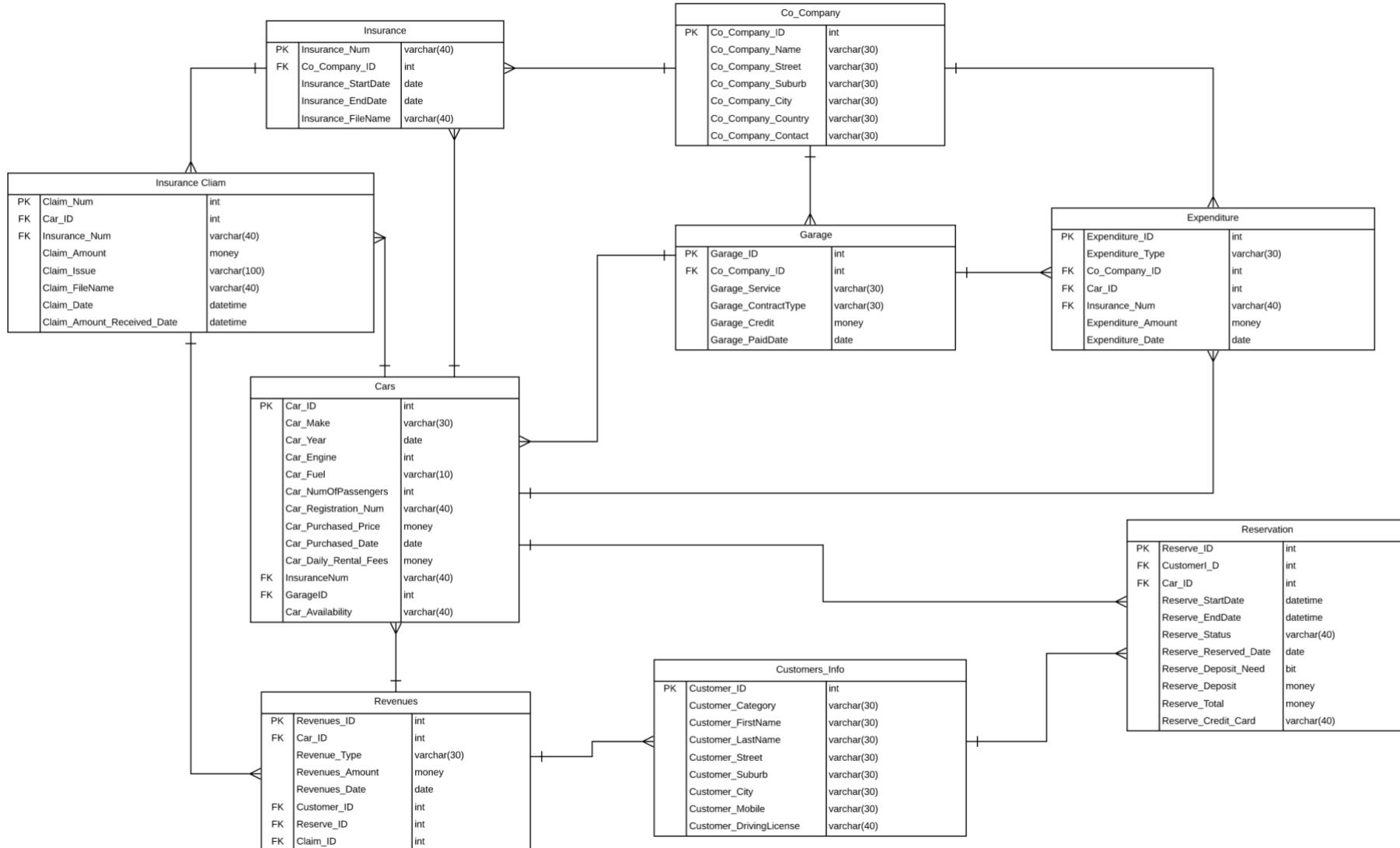
Task	Mark	Result
1. Requirements	10	
2. Logical Design	10	
3. SQL	10	
Sum	30	

Declaration

I declare that this assignment submission will be my own work and I will not collude with anyone else on the preparation of this Assignment.

1. **Requirements:** List and work out the requirements for the database. This will require some imagination and assumption as unfortunately the client brief does not seem to be 100 % complete, though it is reasonably thorough and clear.
 1. All client's details are required to be hold in database.
 2. There are two types of customer group: privileged customers and casual customers.
 3. Privileged customer has the special credit and can make booking in advance for a particular car at any time up to 1 month (assume 1 month = 30 days).
 4. Casual customers are not allowed for the privileges as above, they must pay a deposit for any rental, unless they will pay by credit card.
 5. Assume no customer will exceed the period that they have booked.
 6. Customer needs to sign the desired start date and time and the desired end date and time.
 7. Assume that the customer desired start date and time, cannot be less than 13 hours before the next reservation started on the same car.
 8. Assume that the customer desired end date and time, cannot be less than 1 hours before the next reservation started on the same car.
 9. Assume that customer can cancel their reservation before 24 hours.
 10. All cars details will be kept in database.
 11. All cars will not be held for a period exceeding one year (assume 1 year = 365 days).
 12. Assume that the administrator needs to execute a procedure manually to update the cars' status from available or rental to waiting for sale if meets the condition above (No. 10).
 13. Each car will require an insurance.
 14. Insurance claims must be held on file.
 15. Insurance company details need to be kept in database.
 16. All major repairs and maintenance are done by subcontractors with long-term agreements.
 17. Subcontractor details need to be stored in database.
 18. Agreement of payment is in 2 types: immediately paid after repair, credit allowance for a period.
 19. All purchases need to be registered as company expenditures, regards to repair, maintenance, insurance and so on.
 20. Cash inflow needs to be assigned as company revenues, relates to car hire, car sales and insurance claims.

2. Logical Design: Design the database and produce the Entity Relationship Diagram (ERD).



3. **SQL:** Create the Database using SQL server. You must take screen shot of each step and explain your code.

Part1: Tables

The screenshot shows a SQL query window titled "SQLQuery1.sql - in...ET\2017003399 (59)*". The query itself is:

```
use CAR_Databse;
CREATE TABLE Co_Company
(
    Co_Company_ID int not null,
    Co_Company_Name varchar(30) not null,
    Co_Company_Street varchar(30) not null,
    Co_Company_Suburb VARCHAR(30) not null,
    Co_Company_City varchar(30) not null,
    Co_Company_Country varchar(30) not null,
    Co_Company_Contact varchar(30) not null,
    primary key (Co_Company_ID)
);
```

The "Messages" pane at the bottom right displays the message: "Commands completed successfully."

The “Co_Company” table represents other co-operation companies’ details including **their ID** (primary key), **full name**, **address** and **contact number**.

The screenshot shows a SQL query window in SQL Server Management Studio. The code creates a table named 'Customers_Info' with the following columns:

```
create table Customers_Info
(
    Customer_ID int not null,
    Customer_Category varchar(30) not null,
    Customer_FirstName varchar(30) not null,
    Customer_LastName varchar(30) not null,
    Customer_Street varchar(30) not null,
    Customer_Suburb varchar(30) not null,
    Customer_City varchar(30) not null,
    Customer_Mobile varchar(30) not null,
    Customer_DrivingLicese varchar(40) not null,
    primary key (Customer_ID)
);
```

In the bottom right corner of the window, there is a 'Messages' tab with the status message: "Commands completed successfully."

“Customers_Info” contains all the details about including the **ID** number (primary key), **customer category** (Privileged or Not privileged), **name**, **address**, **mobile** and the **driving license** number.

The screenshot shows a SQL query window titled "SQLQuery1.sql - in...ET\2017003399 (57)*". The code creates a table named "Garage" with the following structure:

```
create table Garage
(
    Garage_ID int not null,
    Co_Company_ID int not null,
    Garage_Service varchar(30) not null,
    Garage_ContractType varchar(30) not null,
    Garage_Credit money null,
    Garage_PaidDate date null,
    primary key (Garage_ID),
    foreign key (Co_Company_ID) references Co_Company (Co_Company_ID)
);
```

In the bottom right corner of the code editor, there is a small yellow icon with a question mark.

Below the code editor, the status bar shows "110 %". In the bottom right corner of the status bar, there is a small yellow icon with a question mark.

The "Messages" tab is selected in the bottom right corner of the interface. It displays the message: "Commands completed successfully."

“Garage” table represents the garage further information including the **a ID number** (primary key), **a co-operation company ID** (foreign key, relates to the Co-company information), the **range of service**, the **contract type** (pay immediately, or pay monthly), the **allowance of monthly credit**, and the **fixed date** that need to pay for repair fees.

The screenshot shows the SQL Server Management Studio interface. The top bar has tabs for 'SQLQuery3.sql - in...ET\2017003399 (61)*' (selected), 'SQLQuery2.sql - in...ET\2017003399 (57)*', and 'SQLQue...'. The main pane displays the following SQL code:

```
create table Insurance
(
    Insurance_Num varchar(40) not null,
    Co_Company_ID int not null,
    Insurance_StartDate date not null,
    Insurance_EndDate date not null,
    Insurance_FileName varchar(40)

    primary key (Insurance_Num),
    foreign key (Co_Company_ID) references Co_Company (Co_Company_ID)
);
```

The code creates a table named 'Insurance' with five columns: 'Insurance_Num' (primary key, varchar(40)), 'Co_Company_ID' (int), 'Insurance_StartDate' (date), 'Insurance_EndDate' (date), and 'Insurance_FileName' (varchar(40)). It includes a primary key constraint on 'Insurance_Num' and a foreign key constraint on 'Co_Company_ID' referencing the 'Co_Company' table.

The bottom pane shows the 'Messages' window with the message: 'Commands completed successfully.'

The “Insurance” table represents the insurance information includes the insurance number (primary key), the Co-operation company information (foreign key, relates to the insurance company details), the start date of this insurance, the end date of this insurance, and the insurance file name that store on the local disk.

The screenshot shows the SQL Server Management Studio interface. In the top tab bar, there are two tabs: "SQLQuery4.sql - in...ET\2017003399 (62)*" and "SQLQuery3.sql - in...ET\2017003399 (61)*". The main area displays the SQL code for creating the "Cars" table. The code defines the table structure with columns for Car_ID, Car_Make, Car_Year, Car_Engine, Car_Fuel, Car_NumOfPassengers, Car_Registration_Num, Car_Purchased_Price, Car_Purchased_Date, Car_Daily_Rental_Fees, Insurance_Num, Garage_ID, and Car_Availability. It includes primary key and foreign key constraints referencing the "Insurance" and "Garage" tables. The code ends with a closing parenthesis and a semicolon. Below the code, the status bar shows "110 %". In the bottom right corner of the interface, there is a "Messages" window with the message "Commands completed successfully."

```
create table Cars
(
    Car_ID int not null,
    Car_Make varchar(30) not null,
    Car_Year date not null,
    Car_Engine int not null,
    Car_Fuel varchar(10)not null,
    Car_NumOfPassengers int not null,
    Car_Registration_Num varchar(30) not null,
    Car_Purchased_Price money not null,
    Car_Purchased_Date date not null,
    Car_Daily_Rental_Fees money not null,
    Insurance_Num varchar(40) not null,
    Garage_ID int not null,
    Car_Availability varchar(40) not null,
    primary key (Car_ID),
    foreign key (Insurance_Num)references Insurance (Insurance_Num),
    foreign key (Garage_ID)references Garage (Garage_ID)
);
```

The “Cars” table represents all the cars details including a unique ID for each car (primary key), the factory that the car made by, the year of the car, the engine size, the fuel type, the maximum number of passengers, the registration number of this car, the price and date that this car was purchased by CAR, the established price for daily rental, the insurance number for this car (foreign key, relates to the insurance details), the garage ID (foreign key, relates to the garage further details), and the availability of this car (waiting for sale, sold, on used, or available).

The screenshot shows two side-by-side SQL Query windows in SQL Server Management Studio.

The left window (SQLQuery7.sql) contains the creation of the `Insurance_Claim` table:

```
create table Insurance_Claim
(
    Claim_Num int not null,
    Car_ID int not null,
    Insurance_Num varchar(40) not null,
    Claim_Amount money not null,
    Claim_Issue varchar(100) not null,
    Claim_FileName varchar(40) not null,
    primary key (Claim_Num),
    foreign key (Car_ID) references Cars (Car_ID),
    foreign key (Insurance_Num) references Insurance (Insurance_Num)
);
```

The right window (SQLQuery6.sql) contains an alter statement for the `Insurance_Claim` table:

```
alter table Insurance_Claim
add Claim_Date datetime;
alter table Insurance_Claim
add Claim_Amount_Received_Date datetime;
```

Both windows show a "Messages" tab at the bottom with the message "Commands completed successfully."

The “Insurance Claim” table represents the information about any insurance been claimed, including the claim ID (primary key), car ID (foreign key, relates to the car information), Insurance number (foreign key, relates to the insurance details), the about of money been claimed, what the issue(s) been claimed, the claim file name related to the relevant document after a claim, the claimed date, and the date that received the fund by insurance company.

```

SQLQuery6.sql - in...ET\2017003399 (64)*  SQLQuery5.sql - in...ET\2017003399 (63)* SQLQuery2.sql - in...ET\2017003399 (57)*  SQLQuery1.sql - in...ET\2017003399 (59)*

create table Reservation
(
    Reserve_ID int not null,
    Customer_ID int not null,
    Car_ID int not null,
    Reserve_StartDate datetime not null,
    Reserve_EndDate datetime not null,
    Reserve_Status varchar(10),
    Reserve_Reserved_Date datetime,
    Reserve_Deposit_Need bit not null DEFAULT 1,
    Reserve_Deposit money null,
    Reserve_Total money null,
    primary key (Reserve_ID),
    foreign key (Customer_ID) references Customers_Info(Customer_ID),
    foreign key (Car_ID) references Cars (Car_ID)
);

alter table Reservation
    alter column Reserve_Status varchar(40);

alter table Reservation
    add Reserve_Credit_Card varchar(40) null;

```

110 %

Messages
Commands completed successfully.

110 %

Messages
Commands completed successfully.

The “Reservation” Table contains a set of information about customers reservations, including the **Reservation ID** (primary key), the **Customer ID** (foreign key, relates to the customer information, especially for checking the customer type to get the privilege or not), the **Car ID** (foreign key, relates to the car information, and shows which car was reserved). The **Start date** for this reservation, the **End date** for this reservation, the **Reservation status** (which can be reserved, reserved with credit card, reserved with deposit, ongoing with deposit, ongoing with credit card, privileged reservation ongoing, cancelled, completed not paid rest, and completed paid), the **date of this reservation issued, whether the customer needs to paid the deposit** based on the customer type in customer table, the **amount of deposit** (can be null value if the customer has privilege), **the total amount** needs to pay, and the credit card number (if applicable, can be null value).

The screenshot shows the SQL Server Management Studio interface with three tabs at the top: 'SQLQuery10.sql - i...ET\2017003399 (65)*', 'SQLQuery9.sql - in...ET\2017003399 (68)*', and 'SQLQuery11.sql - in...ET\2017003399 (69)*'. The central pane displays the SQL code for creating the 'Expenditure' table:

```
create table Expenditure
(
    Expenditure_ID int not null,
    Expenditure_Type varchar(30) not null,
    Co_Company_ID int not null,
    Car_ID int null,
    Garage_ID int null,
    Insurance_Num varchar(40) null,
    Expenditure_Amount money not null,
    Expenditure_Date date not null,

    primary key (Expenditure_ID),
    foreign key (Co_Company_ID) references Co_Company (Co_Company_ID),
    foreign key (Car_ID) references Cars (Car_ID),
    foreign key (Insurance_Num) references Insurance (Insurance_Num),
    foreign key (Garage_ID) references Garage (Garage_ID)
);
```

The bottom pane shows the 'Messages' tab with the status 'Commands completed successfully.'

The “Expenditure” table represents all the information regards to the company cost, including the **Expenditure ID** (primary key), the **type of this expenditure**, the **company ID** that they pay for (foreign key, relates to the co-operation company details), the **car ID** (foreign key, relates to the car information. This column can be null value, if the purchase is not related to car, like the daily consumption of stationary), the **garage ID** (foreign key, relates to the garage information, this is also can be null, if the expenditure is not for repair), the **insurance num** (foreign key, relates to the insurance details, it is can be null value, when the purchase is not about to get or update the insurance for the car), the **amount** of this expenditure, and **the date** of this expenditure.

The screenshot shows the SQL Server Management Studio interface. In the center, there is a code editor window containing the SQL script for creating the 'Revenues' table. The table structure includes columns for Revenues_ID (primary key), Car_ID (foreign key to Customers_Info), Reserve_ID (foreign key to Reservation), and Claim_Num (foreign key to Insurance_Claim). The 'Messages' pane at the bottom indicates that the commands completed successfully.

```
SQLQuery9.sql - in...ET\2017003399 (68))* X SQLQuery8.sql - in...ET\2017003399 (67))* SQLQue
create table Revenues
(
    Revenues_ID int not null,
    Car_ID int not null,
    Revenues_Type varchar(30) not null,
    Revenues_Amount money not null,
    Revenues_Date date not null,
    Customer_ID int null,
    Reserve_ID int null,
    Claim_Num int null,
    primary key (Revenues_ID),
    foreign key (Customer_ID) references Customers_Info (Customer_ID),
    foreign key (Reserve_ID) references Reservation (Reserve_ID),
    foreign key (Claim_Num) references Insurance_Claim (Claim_Num),
);
110 % < Messages
Commands completed successfully.
```

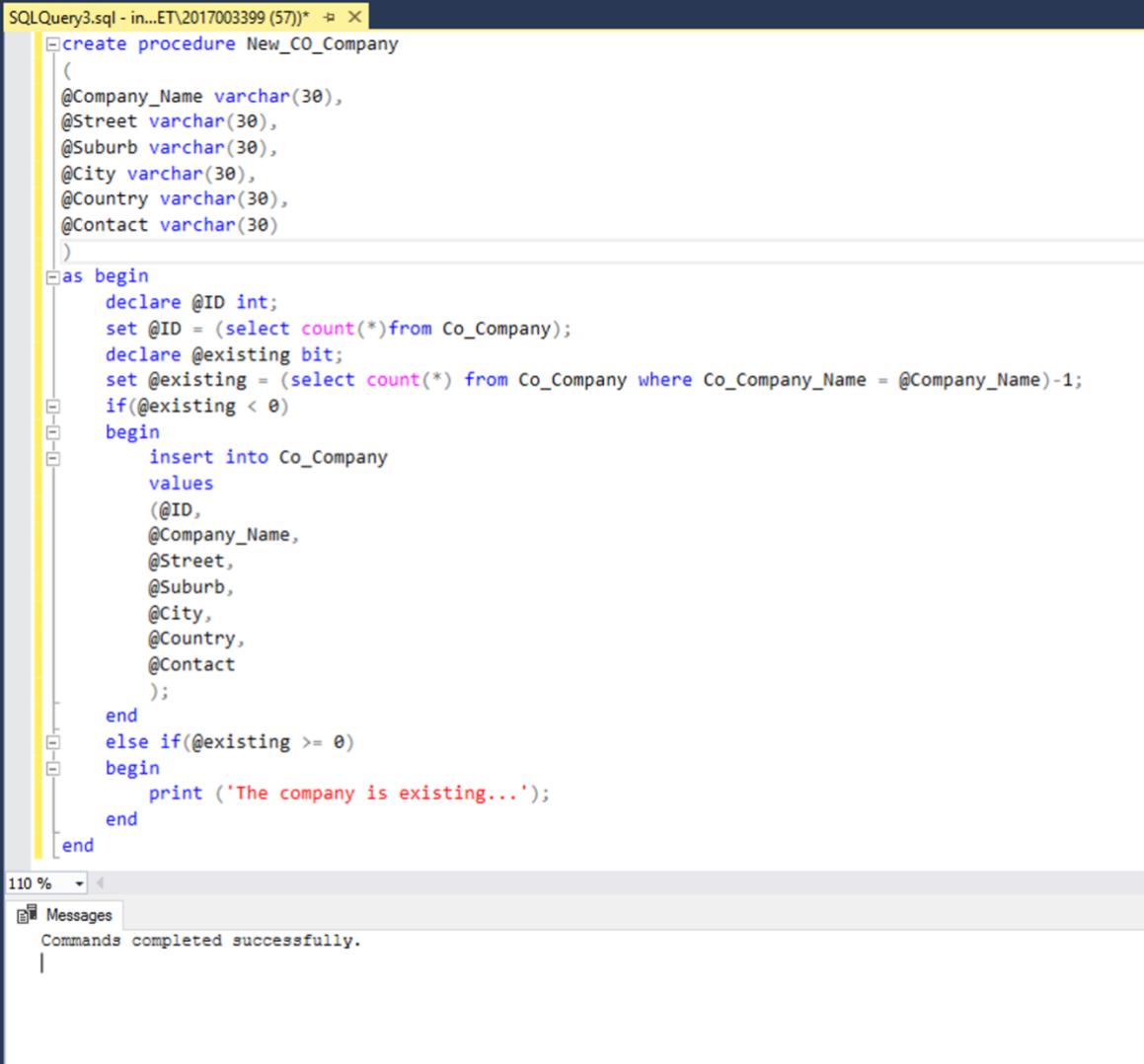
The “Revenue” table relates to the information about company incomes, including the **revenues ID** (primary key), the **car ID** (foreign key, relates to the car information), the **revenue type** (can be rental, reserved rental, car sale, and insurance claim), the **amount** of income received, the **date** of the income received, the **customer ID** (foreign key, relates to customer information, this is can be null value, if the income is insurance claim), **reserve ID** (foreign key, relates to the reservation information, this is can be null value, if the income is not the type of reserved rental), and the **claim number** (foreign key, relates to the insurance claim information. This is can be a null value if the income is not the type of insurance claim).

Part2: Procedures



```
SQLQuery2.sql - in...ET\2017003399 (57)* + X
create procedure New_Customer
(
    @Category varchar(30),
    @FirstName varchar(30),
    @LastName varchar(30),
    @Street varchar(30),
    @Suburb varchar(30),
    @City varchar(30),
    @Mobile varchar(30),
    @DrivingLicense varchar(30)
)
as begin
    declare @ID int;
    set @ID = (select count(*) from Customers_Info);
    insert into Customers_Info values
    (
        @ID,
        @Category,
        @FirstName,
        @LastName,
        @Street,
        @Suburb,
        @City,
        @Mobile,
        @DrivingLicense
    );
    print ('Success..')
end
```

This procedure mainly designed for inserting new customer.



```
SQLQuery3.sql - in...ET\2017003399 (57)* ×
create procedure New_CO_Company
(
    @Company_Name varchar(30),
    @Street varchar(30),
    @Suburb varchar(30),
    @City varchar(30),
    @Country varchar(30),
    @Contact varchar(30)
)
as begin
    declare @ID int;
    set @ID = (select count(*) from Co_Company);
    declare @existing bit;
    set @existing = (select count(*) from Co_Company where Co_Company_Name = @Company_Name)-1;
    if(@existing < 0)
        begin
            insert into Co_Company
            values
            (@ID,
            @Company_Name,
            @Street,
            @Suburb,
            @City,
            @Country,
            @Contact
            );
        end
    else if(@existing >= 0)
        begin
            print ('The company is existing...');
        end
end
110 % ▾
Messages
Commands completed successfully.
```

This procedure just for inserting a new company, it will help user to automatically check the company name if it exists.

The screenshot shows a SQL Server Management Studio (SSMS) window with two tabs at the top: "SQLQuery6.sql - in...ET\2017003399 (59)*" and "SQLQuery4.sql - in...ET\2017003399 (57)*". The "SQLQuery6.sql" tab is active and contains the following T-SQL code:

```
create procedure New_Garage
(
    @Garage_Company_Name varchar(30),
    @Street varchar(30),
    @Suburb varchar(30),
    @City varchar(30),
    @Country varchar(30),
    @Contact varchar(30),
    @Service varchar(30),
    @Contract_Type varchar(30),
    @Credit money,
    @PaidDate date
)
as begin
    declare @Company_existing bit
    set @Company_existing = (select count(*) from Co_Company where @Garage_Company_Name = Co_Company_Name) - 1;
    if(@Company_existing < 0)
        begin
            print ('This company is existing...')
        end
    else if (@Company_existing >= 0)
        begin
            exec New_Co_Company @Garage_Company_Name, @Street, @Suburb, @City, @Country, @Contact;

            declare @Company_ID int;
            set @Company_ID = (select Co_Company_ID from Co_Company where Co_Company_Name = @Garage_Company_Name);
            declare @Garage_ID int;
            set @Garage_ID = (select count(*) from Co_Company) + 1;

            if(@Contract_Type = 'Pay immediately')
                begin
                    insert into Garage values
                    (
                        @Garage_ID,
                        @Company_ID,
                        @Service,
                        @Contract_Type,
                        @Credit,
                        @PaidDate
                    );
                end
        end
end
```

The code creates a stored procedure named "New_Garage" that takes nine parameters. It first checks if a company with the same name already exists in the "Co_Company" table. If not, it executes a stored procedure "New_Co_Company" with the provided parameters. Then, it retrieves the ID of the new company from the "Co_Company" table and inserts a new row into the "Garage" table with the new company ID, the service type, the contract type, credit amount, and payment date.

The "Messages" pane at the bottom shows the message: "Commands completed successfully."

New Garage procedure Part 1

The screenshot shows a SQL Server Management Studio interface with two tabs open:

- SQLQuery6.sql - in...ET\2017003399 (59)***: Contains the stored procedure code.
- SQLQuery4.sql - in...ET\2017003399 (57)***: Contains the command `exec Garage_Proc 'Pay monthly'`.

The stored procedure code is as follows:

```
        @Service,
        @Contract_Type,
        null,
        null
    );
end
else if (@Contract_Type = 'Pay monthly')
begin
insert into Garage values
(
    @Garage_ID,
    @Company_ID,
    @Service,
    @Contract_Type,
    @Credit,
    @PaidDate
);
end
end
end
```

In the bottom right corner of the interface, there is a "Messages" window with the message "Commands completed successfully."

New Garage procedure Part 2

This procedure will help to insert a new garage based on the contract categories which are Immediately pay or pay monthly. Also, it is designed to check if the garage company name is invalid, then will suggest user to insert company information first.

SQLQuery4.sql - in...ET\2017003399 (57)* X

```
create procedure New_Car
(
    @Car_Company_Name varchar(30),
    @Make varchar(30),
    @Year date,
    @Engin int,
    @Fuel varchar(10),
    @Passengers int,
    @Resgitration varchar(30),
    @Purchased_Price money,
    @Purchased_Date date,
    @Daily_Rental_Fees money,
    @Insurance_Num varchar(40),
    @Insurance_Price money,
    @Insurance_Company_Name varchar(40),
    @Insurance_StartDate date,
    @Insurance_EndDate date,
    @Insurance_FileName varchar(40),
    @Garage_ID int
)
as begin
    declare @Garage_valid bit;
    set @Garage_valid = (select count(*) from Garage where Garage_ID = @Garage_ID);

    /*check garage*/
    if(@Garage_valid < 0)
    begin
        print('Invalid Garage ID or Garage is not existing...')
    end
    else if (@Garage_valid >= 0)
    begin
        declare @Insurance_Company_existing bit;
        set @Insurance_Company_existing =
        (select count(*) from Co_Company where Co_Company_Name = @Insurance_Company_Name) - 1;
        /*check insurance company*/
        if(@Insurance_Company_existing < 0)
        begin
            print('Invalid Insurance Company Name')
        end
    end
end
```

110 % ▶

Messages
Commands completed successfully.

Purchase New Car Part 1

The screenshot shows a SQL query window titled "SQLQuery4.sql - in...ET\2017003399 (57)*". The code is a stored procedure that handles the insertion of new insurance company and car records. It includes logic to check if the insurance company exists, insert the insurance record, and then insert the car record.

```
begin
    print ('Unknown insurance company, please insert information of this company first...')
end
else if (@Insurance_Company_existing >= 0)
begin
    declare @Car_Company_exisitng bit;
    set @Car_Company_exisitng = (select count(*) from Co_Company where Co_Company_Name = @Car_Company_Name) - 1;
    if(@Car_Company_exisitng < 0)
    begin
        print ('Unknow Car Company, please complete record in the Co_Company first...')
    end
    else if(@Car_Company_exisitng >= 0)
    begin

        declare @Insurance_Company_ID int;
        set @Insurance_Company_ID = (select Co_Company_ID from Co_Company where Co_Company_Name = @Insurance_Company_Name);

        declare @Car_ID int;
        set @Car_ID = (select count(*) from Cars) + 1;

        /*insert record in Insurance*/
        insert into Insurance values
        (
        @Insurance_Num,
        @Insurance_Company_ID,
        @Insurance_StartDate,
        @Insurance_EndDate,
        @Insurance_FileName
        );
        /*insert record in Cars*/
        insert into Cars values
        (
        @Car_ID,
        @Make,
        @Year,
        @Engin,
        @Fuel,
```

110 %

Messages

Commands completed successfully.

Purchase New Car Part 2

The screenshot shows a SQL query window titled "SQLQuery4.sql - in...ET\2017003399 (57)*". The window contains a large block of SQL code. The code includes declarations for variables like @Fuel, @Passengers, etc., and performs insertions into the Expenditure table. It also includes logic to determine the next ID for the Expenditure table and to insert records for purchasing a car and insurance. The code ends with a successful execution message: "Commands completed successfully."

```
SQLQuery4.sql - in...ET\2017003399 (57)*  X
    @Fuel, |
    @Passengers,
    @Resgitration,
    @Purchased_Price,
    @Purchased_Date,
    @Daily_Rental_Fees,
    @Insurance_Num,
    @Garage_ID,
    'Available'
);
/*insert record in Expenditure*/
declare @E_ID int;
set @E_ID = (select count(*) from Expenditure) + 1;
declare @Car_Company_ID int;
set @Car_Company_ID = (select Co_Company_ID from Co_Company where @Car_Company_Name = Co_Company_Name);
insert into Expenditure values
(
    @E_ID,
    'Purchase car',
    @Car_Company_ID,
    @Car_ID,
    '',
    '',
    '',
    @Purchased_Price,
    @Purchased_Date
);
insert into Expenditure values
(
    @E_ID + 1,
    'Purchase car insurance',
    @Insurance_Company_ID,
    @Car_ID,
    '',
    @Insurance_Num,
    @Insurance_Price,
    @Purchased_Date
);
110 %  <  Messages  Commands completed successfully.
```

Purchase New Car Part 3

The screenshot shows a SQL query window titled "SQLQuery4.sql - in...ET\2017003399 (57)*". The code within the window is a stored procedure definition, likely named "sp_PurchaseNewCarPart4", which includes parameters for insurance price and purchase date, and nested loops or cursors. Below the code, the status bar displays "Commands completed successfully." at 110% zoom.

Purchase New Car Part 4

This procedure will help to check if any information is invalid like insurance company or car factory name, then will recommend user to insert relative information first. If everything correct, then to insert all information related to new car, car's insurance and the garage assigned for car repair, then will generate an expenditure record for this purchase.

The screenshot shows a SQL Server Management Studio window with four tabs at the top: SQLQuery5.sql, SQLQuery4.sql, SQLQuery3.sql, and SQLQuery2.sql. The SQLQuery5.sql tab is active and contains the following T-SQL code:

```
create procedure Repair_Payment
(
    @Garage_Company_Name varchar(30),
    @Garage_ID int,
    @Car_ID int,
    @Amount money,
    @Payment_Date datetime
)
as begin
    declare @Garage_Company_existing int;
    declare @Garage_existing int;
    set @Garage_Company_existing = (select count(*) from Co_Company where @Garage_Company_Name = Co_Company_Name) - 1;
    set @Garage_existing = (select count(*) from Garage where Garage_ID = @Garage_ID) - 1;

    if(@Garage_Company_existing < 0)
    begin
        print ('Cannot find Company information...')
    end
    else if(@Garage_Company_existing = 0)
    begin
        if(@Garage_existing < 0)
        begin
            print ('Cannot find garage information...')
        end
        else if(@Garage_existing = 0)
        begin
            declare @E_ID int;
            set @E_ID = (select count(*) from Expenditure) + 1;
            declare @Company_ID int;
            set @Company_ID = (select Co_Company_ID from Co_Company where @Garage_Company_Name = Co_Company_Name);
            insert into Expenditure values
            (
                @E_ID,
                'Repair Payment',
                @Amount,
                @Payment_Date
            );
        end
    end
end
```

The code is a stored procedure named Repair_Payment. It takes five parameters: @Garage_Company_Name, @Garage_ID, @Car_ID, @Amount, and @Payment_Date. The procedure first checks if the company name exists in the Co_Company table. If it does not, it prints an error message. If the company exists but the garage ID does not, it prints an error message. If both exist but the garage ID does not, it inserts a new row into the Expenditure table with the company ID, a description of 'Repair Payment', the amount, and the payment date.

Repair Payment procedure part 1

The screenshot shows a SQL Server Management Studio window with three tabs at the top: 'SQLQuery5.sql - in...ET\2017003399 (60)*' (active), 'SQLQuery4.sql - in...ET\2017003399 (59)*', and 'SQLQuery3.sql - in...ET'. The main pane displays the following T-SQL code:

```
        'Repair Payment',
        @Company_ID,
        @Car_ID,
        @Garage_ID,
        null,
        @Amount,
        @Payment_Date
    );
end
end
```

In the bottom right corner of the main pane, there is a small yellow status bar with the text '110 %' and a progress bar.

At the bottom of the screen, the 'Messages' tab is selected in the status bar, showing the message: 'Commands completed successfully.'

Repair Payment procedure part 2

This procedure is used for when CAR needs to pay for repair, then to create a new expenditure to sign relevant information in the relative columns.
Also, it is designed to check if garage company does not exist in the Co_Company table, then recommend to insert those information first.

The screenshot shows a SQL query window in SQL Server Management Studio. The title bar reads "SQLQuery8.sql - in...ET\2017003399 (56)*". The code in the window is a stored procedure named "Other_Consumption". The procedure takes four parameters: @Company_Name (varchar(30)), @Items (varchar(30)), @Amout (money), and @DATE (datetime). It first checks if the company exists in the "Co_Company" table. If it does not exist, it prints an error message. If it does exist, it retrieves the company ID and inserts a new row into the "Expenditure" table with the provided parameters and a generated expenditure ID. The status bar at the bottom shows "Commands completed successfully."

```
create procedure Other_Consumption
(
    @Company_Name varchar(30),
    @Items varchar(30),
    @Amout money,
    @DATE datetime
)
as begin
    declare @Company_existing int;
    set @Company_existing = (select count (*) from Co_Company where @Company_Name = Co_Company_Name) - 1;

    if(@Company_existing < 0)
        begin
            print ('Cannot find company information...')
        end
    else if (@Company_existing = 0)
        begin
            declare @Company_ID int;
            set @Company_ID = (select Co_Company_ID from Co_Company where @Company_Name = Co_Company_Name);
            declare @E_ID int;
            set @E_ID = (select count(*) from Expenditure) + 1;

            insert into Expenditure values
            (
                @E_ID,
                @Items,
                @Company_ID,
                null,
                null,
                null,
                @Amout,
                @DATE
            );
        end
end
```

Other consumption procedure

In case of CAR might have some other daily consumptions, so this procedure is designed for this aim.

```
SQLQuery3.sql - in...ET\2017003399 (57)*      SQLQuery2.sql - in...ET\2017003399 (56)*      SQLQuery1.sql - i
create procedure Display_Cars_For_Sale
as begin
    exec Update_Cars_Retired;
    select * from Cars where Car_Availability = 'Waiting for sale';
end
```

110 % < Messages
Commands completed successfully.

Display Cars need to sell

This procedure is designed to show how many cars need to sell, according to the policy of CAR, “All cars will not be hold for a period exceeding one year”.

The screenshot shows a SQL Server Management Studio (SSMS) interface with two tabs open: 'SQLQuery3.sql' and 'SQLQuery2.sql'. The 'SQLQuery3.sql' tab contains the following T-SQL code:

```
create procedure Update_Cars_Retired
as begin
update Cars
set Car_Availability = 'Waiting for sale'
where DATEDIFF(day,Car_Purchased_Date, GETDATE()) >= 365;
end
```

The 'Messages' pane at the bottom right displays the message: 'Commands completed successfully.'

This procedure aims for supporting the procedure above, to update all cars need to sell. (days on hold over 364 days)

The screenshot shows a SQL Server Management Studio (SSMS) window with two tabs: 'SQLQuery3.sql' and 'SQLQuery2.sql'. The 'SQLQuery3.sql' tab contains the following T-SQL code:

```
create procedure Sell_Cars
(
    @Car_ID int,
    @Amount money
)
as begin
    update Cars
    set Car_Availability = 'Sold'
    where Car_ID = @Car_ID;

    declare @R_ID int;
    set @R_ID = (select count(*) from Revenues) + 1;
    insert into Revenues values
    (
        @R_ID,
        @Car_ID,
        'Car sale',
        @Amount,
        GETDATE(),
        null,
        null,
        null
    );
end
```

The 'Messages' pane at the bottom right of the SSMS window displays the message: "Commands completed successfully."

When the car has been sold, user needs to insert the car ID and amount of money been received to this procedure, then it will automatically update the car status to sold and create a new revenue record with relevant information.

The screenshot shows the SQL Server Management Studio interface with four tabs at the top: SQLQuery7.sql, SQLQuery6.sql, SQLQuery5.sql, and SQLQuery4.sql. The SQLQuery4.sql tab is active, displaying the following T-SQL code:

```
create procedure Insurance_Claim_Procedure
(
    @Insurance_Num varchar(40),
    @Insurance_Company_Name varchar(30),
    @Car_ID int,
    @Claim_Num int,
    @Amount_Claimed money,
    @Issues varchar(100),
    @Claim_FileName varchar(40),
    @Claim_Date datetime,
    @Amount_Received_Date datetime
)
as begin
    declare @Car_existing int;
    set @Car_existing = (select count(*) from Cars where @Car_ID = Car_ID) - 1;
    declare @Insurance_Company_existing int;
    set @Insurance_Company_existing = (select count(*) from Co_Company where @Insurance_Company_Name = Co_Company_Name) - 1;
    declare @Insurance_existing int;
    set @Insurance_existing = (select count(*) from Insurance where @Insurance_Num = Insurance_Num) - 1;

    /* check the insurance company is valid */
    if( @Insurance_Company_existing < 0)
    begin
        print ('Cannot find insurance company record...')
    end
    else if(@Insurance_Company_existing = 0)
    begin
        /* check the car information is available */
        if(@Car_existing < 0)
        begin
            print ('Cannot find car record...')
        end
        else if(@Car_existing = 0)
        begin
            /* check the insurance information is available */
            if(@Insurance_existing < 0)
            begin
                print ('Cannot find insurance record...')
            end
        end
    end
end
```

In the bottom left corner of the code editor, there is a "Messages" pane showing the message: "Commands completed successfully."

At the bottom of the screen, the status bar displays: "inv-vdi-bitsal1 (13.0 SP2) SITNET\2017003399 (59) CAR Database 00:00:00 0 rows".

Insurance Claim procedure Part 1

The screenshot shows a SQL Server Management Studio window with four tabs at the top: SQLQuery7.sql, SQLQuery6.sql, SQLQuery5.sql, and SQLQuery4.sql (the active tab). The code in SQLQuery4.sql is a stored procedure:begin
 print ('Cannot find Insurance record...')
end
else if(@Insurance_existing = 0)
begin
 /* create new claim record */
 insert into Insurance_Claim values
 (
 @Claim_Num,
 @Car_ID,
 @Insurance_Num,
 @Amount_Claimed,
 @Issues,
 @Claim_FileName,
 @Claim_Date,
 @Amount_Received_Date
);

 /* create new revenue record */
 declare @Revenue_ID int;
 set @Revenue_ID = (select count(*) from Revenues) + 1;

 insert into Revenues values
 (
 @Revenue_ID,
 @Car_ID,
 'Insurance claim',
 @Amount_Claimed,
 @Amount_Received_Date,
 null,
 null,
 @Claim_Num
);
end
end
end

The Messages pane at the bottom shows "Commands completed successfully." and the status bar indicates "Query executed successfully." and the connection details.

Insurance claim procedure Part 2

This procedure is for inserting a new insurance claim record in the Insurance Claim table and generate a new revenue record in Revenue table.

The screenshot shows a SQL Server Management Studio window. At the top, there are four tabs: 'SQLQuery9.sql - in...ET\2017003399 (62)*', 'SQLQuery8.sql - in...ET\2017003399 (57)*' (which is highlighted in yellow), 'SQLQuery7.sql - in...ET\2017003399 (61)*', and 'SQLQuery6.sql - in...ET\2017003399 (56)*'. The main pane displays the following T-SQL code:

```
create procedure Display_Cars_Availability
as begin
    /*Update car availability if the car has been reserved, and the reservation will be started within 60 minutes*/
    update Cars
    set Car_Availability = 'Reserved'
    where Car_ID =
        (select Car_ID from Reservation where datediff(MINUTE,getdate(),Reserve_StartDate) < 60);
    /*Update car availability if the car exceeds 365 days since CAR purchased*/
    exec Update_Cars_Retired;
    /*display the cars available for booking*/
    select Car_ID,Car_Make,Car_Year, Car_Engine, Car_Fuel, Car_NumOfPassengers
    , Car_Daily_Rental_Fees, Car_Availability from Cars
    where Car_Availability = 'Available'
end
```

Below the code, the status bar shows '110 %' and a 'Messages' button. The 'Messages' section contains the message: 'Commands completed successfully.'

Booking system 1 – Display Cars Availability

It will update every car to “Reserved” if it has reservation which will be started within 60 minutes, then display the cars are available for rental.

The screenshot shows a SQL Server Management Studio (SSMS) window with two tabs open. The top tab is titled 'SQLQuery7.sql - in...ET\2017003399 (61)*' and contains the following T-SQL code:

```
create procedure Check_Car_Duration_Availability
(
    @Car_ID int,
    @StartDate datetime,
    @EndDate datetime
)
as begin

    declare @Car_existing int;
    set @Car_existing = (select count(*) from Cars where Car_Availability = 'Available') - 1;

    if(@Car_existing < 0)
    begin
        print ('The car you picked, is not available or not existing...')
    end
    else if(@Car_existing = 0)
    begin
        /*check if the car has been reserved or not*/
        declare @StartDate_has_Reserved int;
        set @StartDate_has_Reserved = (select count(*) from Reservation where Car_ID = @Car_ID
            and Reserve_Status != 'Cancelled' and Reserve_Status != 'Completed paid'
            and Reserve_Status != 'Completed paid' and Reserve_Status != 'Completed not paid'
            and Reserve_Status != 'Completed not paid rest' and Reserve_Status != 'Completed not charge from Credit Card'
            and Reserve_StartDate > GETDATE());

        /*no any reservation record*/
        if(@StartDate_has_Reserved < 0)
        begin
            print ('This car is available, no body reserved it...')
        end
        /*has reservation records*/
        else if(@StartDate_has_Reserved > 0)
        begin
            /*get the number of hours between the desired start date to the nearest reservation start date on same car*/
            declare @StartDate_Available int;
            set @StartDate_Available = datediff(hour,@StartDate,
```

The bottom tab is titled 'SQLQuery6.sql - in...ET\2017003399 (56)*'.

In the SSMS interface, there is a 'Messages' pane at the bottom left showing the message: 'Commands completed successfully.'

Booking system 2 – Check Desired Rental duration availability of the car Part 1

The screenshot shows a SQL Server Management Studio window. The top tab bar has two tabs: "SQLQuery7.sql - in...ET\2017003399 (61)*" and "SQLQuery6.sql - in...ET\2017003399 (56)*". The main pane displays a T-SQL script for a stored procedure. The script uses dynamic SQL to find the nearest reservation start date for a given car ID. It then checks if the desired start date is less than 13 hours away from the nearest reservation. If so, it prints a message indicating the car is not available. Otherwise, it prints the reserved start date and the date one hour before it. The script ends with several nested end statements. The bottom pane shows the "Messages" tab with the status "Commands completed successfully." and a zoom level of 110%.

```
(select Top 1 Reserve_StartDate from Reservation
 where Car_ID = @Car_ID order by Reserve_Reserved_Date desc));
/*get the nearest reservation start date*/
declare @Reserved_StartDate datetime;
set @Reserved_StartDate = (select Top 1 Reserve_StartDate from Reservation
 where Car_ID = @Car_ID order by Reserve_Reserved_Date desc);

if(@StartDate_Avialable < 13)
begin
    print ('This car is not available for you, due to less than 13|
hours for next reservation before your desired End date and time')
    print ('It has been reserved at:')
    print (@Reserved_StartDate)
end
else if (@StartDate_Avialable >= 13)
begin
    /*the customer desired end date and time,
cannot be less than 1 hours before the next
reservation started on the same car.*/
    print ('This car is available before:')
    print (dateadd(hour,-1,@Reserved_StartDate));
end
end
end
end
```

Booking system 3 – Check desired rental duration availability of the car Part 2

This procedure will help customer to check whether their desired period of time for car rental is suitable for the car itself, and also can tell when will the nearest reservation started.

The screenshot shows a SQL Server Management Studio window with four tabs at the top: SQLQuery5.sql, SQLQuery4.sql, SQLQuery3.sql, and SQLQuery2.sql. The main pane displays a T-SQL stored procedure named 'Booking'. The code performs several checks: it declares variables for customer ID, car ID, and reserve start/end dates; it checks car availability by selecting from the 'Cars' table; it checks customer category by selecting from the 'Customers_Info' table; it calculates the total number of days between reserve dates; it checks if the car is available ('Available') or not; if available, it calculates total rental fees; it generates a reservation ID by counting existing reservations and adding 1; it checks if the customer is 'Privileged'; if so, it checks if the reserved date is more than 30 days from the current date; if valid, it inserts a new record into the 'Reservation' table. The status bar at the bottom shows '110 %' zoom, 'Messages' tab selected, and the message 'Commands completed successfully.'

```
create procedure Booking
(
    @Customer_ID int,
    @Car_ID int,
    @Reserve_StartDate datetime,
    @Reserve_EndDate datetime
)
as begin
    declare @Car_Availability varchar(40);
    set @Car_Availability = (select Car_Availability from Cars where Car_ID = @Car_ID);
    declare @Customer_Category varchar(30);
    set @Customer_Category = (select Customer_Category from Customers_Info where Customer_ID = @Customer_ID);

    declare @Total_Days int;
    set @Total_Days = datediff(day, @Reserve_StartDate,@Reserve_EndDate);

    /* check car availability */
    if(@Car_Availability != 'Available')
    begin
        print 'The car you selected, is not available at moment...'
    end

    else if (@Car_Availability = 'Available')
    begin
        /*Check Customer category*/
        declare @Total money;
        set @Total = (select Car_Daily_Rental_Fees * @Total_Days from Cars where Car_ID = @Car_ID);

        declare @Reserve_ID int;
        set @Reserve_ID = (select count(*) from Reservation) + 1;

        if (@Customer_Category = 'Privileged')
        begin
            /*Check Reserved Date*/
            if(datediff(day,@Reserve_StartDate, GETDATE()) !> 30)
            begin
                insert into Reservation values
    
```

Booking system 4 – Booking procedure Part 1

The screenshot shows a SQL Server Management Studio window with a vertical yellow scrollbar on the left. The main pane displays a T-SQL script for a stored procedure. The script includes logic to check if a reservation can be made before 30 days, print an error message if it is, and then insert a record into the Reservation table if the condition is not met. It also checks the customer category and reserved date. The status bar at the bottom shows 'Commands completed successfully.'

```
SQLQuery5.sql - in...ET\2017003399 (52)* -> X SQLQuery4.sql - in...ET\2017003399 (63))* SQLQuery3.sql - in...ET\2017003399 (61))* SQLQuery2.sql - in...ET\2017003399 (60)*

(
    @Reserve_ID,
    @Customer_ID,
    @Car_ID,
    @Reserve_StartDate,
    @Reserve_EndDate,
    'Reserved',
    GETDATE(),
    0,
    0,
    @Total,
    null
);
end
else if (datediff(day,@Reserve_StartDate,GETDATE()) >= 30)
begin
    print ('Sorry, you cannot reserve a car before more than 30 days, including 30 days')
end
end

/*Check Customer Type*/

else if (@Customer_Category = 'Not Privileged')
begin
/*Check Reserved Date*/
    if(datediff(day,@Reserve_StartDate, GETDATE()) !> 30)
    begin
        insert into Reservation values
        (
            @Reserve_ID,
            @Customer_ID,
            @Car_ID,
            @Reserve_StartDate,
            @Reserve_EndDate,
            'Awaiting Deposit',
            GETDATE(),
            0,
            0
        );
    end
end
else
begin
    print ('Sorry, you cannot reserve a car before more than 30 days, including 30 days')
end
end

110 % < >
Messages Commands completed successfully.
```

Booking system 5 – Booking procedure Part 2

The screenshot shows a SQL Server Management Studio window with four tabs at the top: 'SQLQuery5.sql - in...ET\2017003399 (52)*' (selected), 'SQLQuery4.sql - in...ET\2017003399 (63)*', 'SQLQuery3.sql - in...ET\2017003399 (61)*', and 'SQLQuery2.sql - in...ET\2017003399 (60)*'. The main pane displays a T-SQL script for a stored procedure. The script includes several 'print' statements and logic to check if a reservation has been completed or if it's over 30 days from the start date. The script ends with two 'end' keywords. Below the script, the status bar shows '110 %' and 'Messages'. The 'Messages' tab is selected, displaying the message 'Commands completed successfully.'

Booking system 6 – Booking procedure Part 3

This procedure will interact with Reservation table with the customer input. It will create a new reservation record based on ‘customerID’ which can help to find out the customer category in ‘Customer_Info’ table. If the customer is privileged, then will check the reservation start date if over 30 days from the current date which CAR does not allow privileged user to book a car before 30 days. Or if the customer is not privileged, then will sign the reservation status as ‘waiting deposit’, then will recommend to jump to deposit procedure.

```
create procedure Deposit_Procedure
(
    @Reservation_ID int,
    @Option varchar(40),
    @Amount money,
    @Credit_Card varchar(40)
)
as begin
    if(@Option = 'Deposit' and @Option != 'Credit Card')
        begin
            update Reservation
            set Reserve_Status = 'Reserved with Deposit', Reserve_Deposit = @Amount
            where Reserve_ID = @Reservation_ID;

            Declare @Customer_ID int;
            declare @Car_ID int;
            set @Customer_ID = (select Customer_ID from Reservation where Reserve_ID = @Reservation_ID);
            set @Car_ID = (select Car_ID from Reservation where Reserve_ID = @Reservation_ID);

            declare @id int;
            set @id = (select count(*)from Revenues);
            insert into Revenues values (@id+1, @Car_ID, 'Rental Deposit', @Amount, getdate(), @Customer_ID, @Reservation_ID, null);
        end
    else if (@Option = 'Credit Card' and @Option != 'Deposit')
        begin
            update Reservation
            set Reserve_Status = 'Reserved with Credit Card', Reserve_Credit_Card = @Credit_Card
            where Reserve_ID = @Reservation_ID;
        end
    end
end
```

Booking system 7 – Deposit procedure

Once the CAR received the customer option which can be ‘Place deposit’ with the amount of deposit or ‘Provide credit card number’ with the credit card number, then the procedure will fill in the information based on the options. And finally, change the reservation status to either ‘Reserved with Deposit’ or ‘Reserved with Credit Card’. If customer pays the deposit on the reservation, procedure will automatically generate a revenue record in Revenue table.

The screenshot shows a SQL query editor window titled "SQLQuery6.sql - in...ET\2017003399 (56)*". The code displays a stored procedure named "Cancel Reserve" which takes a parameter "@Reserve_ID int". The procedure begins by checking the hours left before the reservation started using the formula `(select datediff(hour, getdate(), Reserve_StartDate) from Reservation where Reserve_ID = @Reserve_ID)`. If the availability is greater than 24 hours, it retrieves the reservation status and amount from the "Reservation" table. It then checks if the status is "Reserved with Deposit". If so, it declares variables for Car_ID, Customer_ID, and E_ID, and inserts a new row into the "Expenditure" table with the status "Reservation cancelled", the car ID, the amount, and the current date.

```
create procedure Cancel Reserve
(
    @Reserve_ID int
)
as begin

    /*check the hours left before the reservation started*/
    declare @availability int;
    set @availability = (select datediff(hour, getdate(), Reserve_StartDate)
    from Reservation where Reserve_ID = @Reserve_ID);

    if(@availability > 24)
    begin
        declare @Reserve_Status varchar(40);
        set @Reserve_Status = (select Reserve_Status from Reservation where Reserve_ID = @Reserve_ID);
        declare @Amount money;
        set @Amount = (select Reserve_Deposit from Reservation where @Reserve_ID = Reserve_ID);

        if(@Reserve_Status = 'Reserved with Deposit')
        begin
            declare @Car_ID int;
            declare @Customer_ID int;
            set @Car_ID = (select Car_ID from Reservation where Reserve_ID = @Reserve_ID);
            set @Customer_ID = (select Customer_ID from Reservation where Reserve_ID = @Reserve_ID);
            declare @E_ID int;
            set @E_ID = (select count(*) from Expenditure) +1;

            insert into Expenditure values
            (
                @E_ID,
                'Reservation cancelled',
                '',
                @Car_ID,
                '',
                '',
                @Amount,
                getdate()
            );
        end
    end
end
```

Messages
Commands completed successfully.

Reservation Cancellation Part 1

The screenshot shows a SQL Server Management Studio window. The title bar reads "SQLQuery6.sql - in...ET\2017003399 (56)*". The main pane contains the following T-SQL code:

```
update Reservation
set Reserve_Status = 'Cancelled'
where Reserve_ID = @Reserve_ID
end
else if (@availability <= 24)
begin
    print ('Cancellation failed. Your Reservation is less than 24 hours...')
end
end
```

The status bar at the bottom indicates "110 %". Below the status bar, the "Messages" tab is selected, showing the message "Commands completed successfully."

Reservation Cancellation Part 2

This procedure will allow customers to cancel their reservation only before 24 hours. If it can be cancelled and also reserved with deposit, it will generate an new Expenditure record with the Expenditure type 'Reservation cancelled'.

The screenshot shows a SQL Server Management Studio window with four tabs at the top: SQLQuery5.sql, SQLQuery4.sql, SQLQuery3.sql, and SQLQuery2.sql. The SQLQuery5.sql tab is active and contains the following T-SQL code:

```
create procedure Customer_Pick_Car
(
    @Reserve_ID int
)
as begin
    declare @reserve_existing int;
    set @reserve_existing = (select count(*) from Reservation where @Reserve_ID = Reserve_ID) - 1;

    if(@reserve_existing < 0)
        begin
            print ('Invalid reservation ID...')
        end
    else if (@reserve_existing = 0)
        begin
            declare @Reserved_Status varchar(40);
            set @Reserved_Status = (select Reserve_Status from Reservation where @Reserve_ID = Reserve_ID);

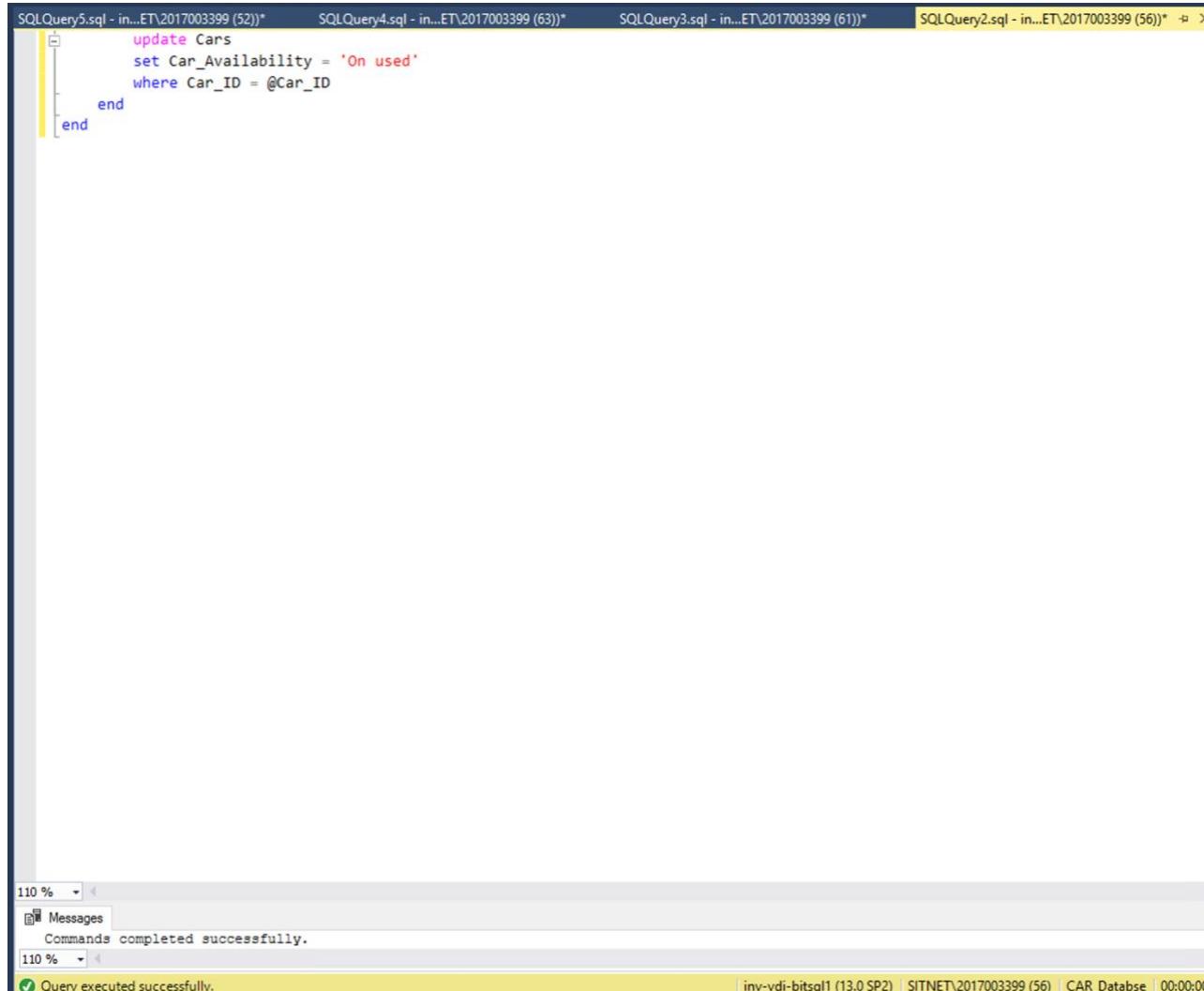
            if(@Reserved_Status = 'Reserved')
                begin
                    update Reservation
                    set Reserve_Status = 'Privileged Reservation Ongoing'
                    where Reserve_ID = @Reserve_ID;
                end
            else if(@Reserved_Status = 'Reserved with Deposit')
                begin
                    update Reservation
                    set Reserve_Status = 'Ongoing with Deposit'
                    where Reserve_ID = @Reserve_ID;
                end
            else if (@Reserved_Status = 'Reserved with Credit Card')
                begin
                    update Reservation
                    set Reserve_Status = 'Ongoing with Credit Card'
                    where Reserve_ID = @Reserve_ID;
                end

            declare @Car_ID int;
            set @Car_ID = (select Car_ID from Reservation where Reserve_ID = @Reserve_ID);

            update Cars
        end
    end
end
```

In the bottom pane, there is a 'Messages' window with the message: 'Commands completed successfully.'

A procedure for customers when they come to pick the car with the reservation ID – Part 1



The screenshot shows a SQL Server Management Studio window with four tabs at the top: SQLQuery5.sql, SQLQuery4.sql, SQLQuery3.sql, and SQLQuery2.sql. The SQLQuery2.sql tab is active, displaying the following T-SQL code:

```
update Cars
set Car_Availability = 'On used'
where Car_ID = @Car_ID
end
end
```

Below the tabs, there is a large empty workspace. At the bottom of the window, the Messages pane shows:

- Commands completed successfully.
- Query executed successfully.

The status bar at the bottom right indicates the session details: inv-vdi-bitsql1 (13.0 SP2) | SITNET\2017003399 (56) | CAR_Database | 00:00:00.

A procedure for customers when they come to pick the car with the reservation ID – Part 2

Just need to insert Reservation ID, then the procedure will automatically validate the reservation ID, if valid, then will update all relative information regards to Car's status and Reservation status.

The screenshot shows the SQL Server Management Studio interface with four tabs at the top: SQLQuery4.sql, SQLQuery3.sql, SQLQuery2.sql, and SQLQuery1.sql. The SQLQuery4.sql tab is active, displaying the following T-SQL code:

```
create procedure Customer_Return_Car
(
    @Reserve_ID int
)
as begin
    declare @reserve_existing int;
    set @reserve_existing = (select count(*) from Reservation where @Reserve_ID = Reserve_ID) - 1;

    if(@reserve_existing < 0)
        begin
            print ('Invalid reservation ID...')
        end
    else if (@reserve_existing = 0)
        begin
            declare @Reserved_Status varchar(40);
            set @Reserved_Status = (select Reserve_Status from Reservation where @Reserve_ID = Reserve_ID);

            if(@Reserved_Status = 'Privileged Reservation Ongoing')
                begin
                    update Reservation
                    set Reserve_Status = 'Completed not paid'
                    where Reserve_ID = @Reserve_ID;
                end
            else if(@Reserved_Status = 'Ongoing with Deposit')
                begin
                    update Reservation
                    set Reserve_Status = 'Completed not paid rest'
                    where Reserve_ID = @Reserve_ID;
                end
            else if (@Reserved_Status = 'Ongoing with Credit Cards')
                begin
                    update Reservation
                    set Reserve_Status = 'Completed not charge from Credit Card'
                    where Reserve_ID = @Reserve_ID;
                end

            select (Reserve_Total - Reserve_Deposit) as Need_to_pay from Reservation where Reserve_ID = @Reserve_ID;
        end
    end
end
```

In the Messages pane at the bottom, it says "Commands completed successfully."

Customer Return a car

When customer return a car , this procedure will update the reservation status to ‘completed not paid’ , ‘completed not paid rest’ or ‘completed not charge form credit card’ based on reservation ID, and the reservation type.

The screenshot shows a SQL Server Management Studio (SSMS) window with four tabs at the top: SQLQuery5.sql, SQLQuery4.sql, SQLQuery3.sql, and SQLQuery2.sql. The SQLQuery4.sql tab is active, displaying the following T-SQL code:

```
create procedure After_Received_Fees
(
    @Reserve_ID int,
    @Amount_Paid money
)
as begin
    declare @reserve_existing int;
    set @reserve_existing = (select count(*) from Reservation where @Reserve_ID = Reserve_ID) - 1;

    if(@reserve_existing < 0)
    begin
        print ('Invalid reservation ID...')
    end
    else if (@reserve_existing = 0)
    begin
        declare @Reserved_Status varchar(40);
        set @Reserved_Status = (select Reserve_Status from Reservation where @Reserve_ID = Reserve_ID);
        declare @Amount_Need_Pay money;
        set @Amount_Need_Pay = (select (Reserve_Total - Reserve_Deposit) from Reservation where @Reserve_ID = Reserve_ID);
        declare @Car_ID int;
        set @Car_ID = (select Car_ID from Reservation where Reserve_ID = @Reserve_ID);
        declare @Customer_ID int;
        set @Customer_ID = (select Customer_ID from Reservation where Reserve_ID = @Reserve_ID);

        if(@Amount_Need_Pay > @Amount_Paid)
        begin
            declare @rest_amount money;
            set @rest_amount = (select (Reserve_Total - Reserve_Deposit - @Amount_Paid) as Need_to_pay
                                from Reservation where Reserve_ID = @Reserve_ID);
            print ('Your still need to pay:')
            print (@rest_amount)

            update Reservation
            set Reserve_Deposit += @Amount_Paid
            where Reserve_ID = @Reserve_ID;

            declare @R_ID1 int;
        end
    end
end
```

In the bottom status bar, it says "Commands completed successfully." and "Acti".

CAR Received Fees Part 1

The screenshot shows a SQL Server Management Studio window with multiple tabs at the top: SQLQuery5.sql, SQLQuery4.sql, SQLQuery3.sql, and SQLQuery2.sql. The main pane displays a T-SQL script for a stored procedure. The script begins by setting a variable @R_ID1 to the count of rows in the Revenues table plus one. It then inserts a new row into the Revenues table with values: @R_ID1, @Car_ID, 'Reserved Rental', @Amount_Paid, GETDATE(), @Customer_ID, @Reserve_ID, and null. The script then checks if the amount paid equals the amount needed to pay. If true, it updates the Reservation table to set Reserve_Status to 'Completed paid' where Reserve_ID equals @Reserve_ID. It then checks if the reserved status is 'Completed not paid rest'. If true, it updates the Reservation table to set Reserve_Status to 'Completed paid' where Reserve_ID equals @Reserve_ID. Finally, it checks if the reserved status is 'Completed not charge from Credit Cards'. If true, it updates the Reservation table to set Reserve_Status to 'Completed paid' where Reserve_ID equals @Reserve_ID. The script concludes by selecting the Customer_ID from the Reservation table where Reserve_ID equals @Reserve_ID, declaring a variable @R_ID2, and setting it to the count of rows in the Revenues table plus one. A message at the bottom of the results pane states 'Commands completed successfully.'

```
SQLQuery5.sql - in...ET\2017003399 (52)* SQLQuery4.sql - in...ET\2017003399 (63)* × SQLQuery3.sql - in...ET\2017003399 (61)* SQLQuery2.sql - in...ET\2017003399 (56)*
set @R_ID1 = (select count(*) from Revenues) + 1;

insert into Revenues values
(
@R_ID1,
@Car_ID,
'Reserved Rental',
@Amount_Paid,
GETDATE(),
@Customer_ID,
@Reserve_ID,
null
);
end
else if (@Amount_Need_Pay = @Amount_Paid)
begin
if(@Reserved_Status = 'Completed not paid')
begin
update Reservation
set Reserve_Status = 'Completed paid'
where Reserve_ID = @Reserve_ID;
end
else if(@Reserved_Status = 'Completed not paid rest')
begin
update Reservation
set Reserve_Status = 'Completed paid'
where Reserve_ID = @Reserve_ID;
end
else if (@Reserved_Status = 'Completed not charge from Credit Cards')
begin
update Reservation
set Reserve_Status = 'Completed paid'
where Reserve_ID = @Reserve_ID;
end
set @Customer_ID = (select Customer_ID from Reservation where Reserve_ID = @Reserve_ID);
declare @R_ID2 int;
set @R_ID2 = (select count(*) from Revenues) + 1;

```

CAR Received Fees Part 2

The screenshot shows a SQL Server Management Studio window with four tabs at the top: SQLQuery5.sql, SQLQuery4.sql, SQLQuery3.sql, and SQLQuery2.sql. The SQLQuery4.sql tab is active, displaying a stored procedure script:insert into Revenues values
(
@R_ID2,
@Car_ID,
'Reserved Rental',
@Amount_Paid,
GETDATE(),
@Customer_ID,
@Reserve_ID,
null
);

update Cars
set Car_Availability = 'Available'
where Car_ID = @Car_ID
end
end

In the Messages pane below, it says "Commands completed successfully." In the status bar at the bottom, it says "Query executed successfully." and "0 rows".

CAR Received Fees Part 3

Once the CAR received the amount of rental fees, procedure will check if the reservation ID is invalid, then check whether the customer pay enough fees, and update the reservation status to completed, and create a new revenue record in Revenue table.

