ICS 220 – Programming Fundamentals
Assignment 3

Scenario:

The B&M company sells cars. They want to create a software system that manages their employees and the cars that they sell. A part of their software requirement is given below. You are required to do a selfstudy on how the car sales function and create a design for managing the functions of the company. In addition to the requirements below, you are encouraged to add more attributes and functionalities to the system. The company has different employees who are either Mangers or Salespersons. An example of a partial file of employees is given below. The employees' personal details like their age, date of birth, and passport details are also stored in the system. Each manager has a group of salespersons assigned to work with her/him. For each sale a salesperson makes, 6.5% of the profit goes to the salesperson, 3.5% goes to the manager, and the rest to the company. The profit is the amount that the salesperson makes above the price of the car.

Name	ID Number	Department	Job Title	Basic Salary
Susan Meyers	47899	Accounting	Manager	37,500
Mark Jones	39119	IT	Salesperson	26,000
Joy Rogers	81774	Manufacturing	Salesperson	24,000

The company has three types of cars, hatchback, sedan, and SUV. An example of a partial file of cars is given below. Other details of each car, like the fuel capacity, max speed, and color are also stored in the system.

Name	ID Number	Price	Type
Jazz	VX3	55,000	Hatch
Mark3	SX3	84,000	Sedan
Wagoner	ZX3	125,000	SUV

Your program must have the following functions:

1. System should be able to add/delete/modify details of employees and cars with GUI.
2. Display all the details of an employee, given the ID number.
3. Display all the details of a car, given the ID number.
4. Display sales details of an employee, given the ID number
5. All details must be stored in binary files using the Pickle library in Python.
6. Any wrong or exceptional input must be appropriately handled.
7. From the above example, assume that Joy Rogers and Mark Jones are supervised by Susan Meyers i.e. Susan is their manager. Mark sells 2 hatchback cars and 3 SUVs this month. Joy sells 1 SUV, 2 hatchbacks, and 2 sedans this month. Your program should show the salaries that all three of them would get this month after they make the sale.

Requirements

1. Design a UML class diagram representing the concepts and relationships in the scenario. Ensure using the different types of association and inheritance relationships where necessary. You may make assumptions about attributes (with proper access specified) and concepts not explicitly mentioned in the problem statement. A clear description of the relationships and assumptions must be included.
2. Write Python code to implement your UML diagram. Ensure that you define test cases to showcase the program features.
3. Ensure that your UML diagram and the Python code are well-documented and structured.

Submission:

1. Submit a report (single PDF file) that has the following sections:
    a. UML Class Diagram and Description
    b. Python classes (copy-paste the code, NOT an image of the code)
        i. The code must be well documented with good coding standards followed.
    c. Github repository link, with access, made public:
        i. Ensure to include the gradual progress of your work in the Github repository.
        ii. The repository would indicate the cumulative progress of your work in the assignment.
    d. Summary of learnings
2. Submission deadline: 10/May/2023, 11:59 PM

Learning Outcomes Assessed:

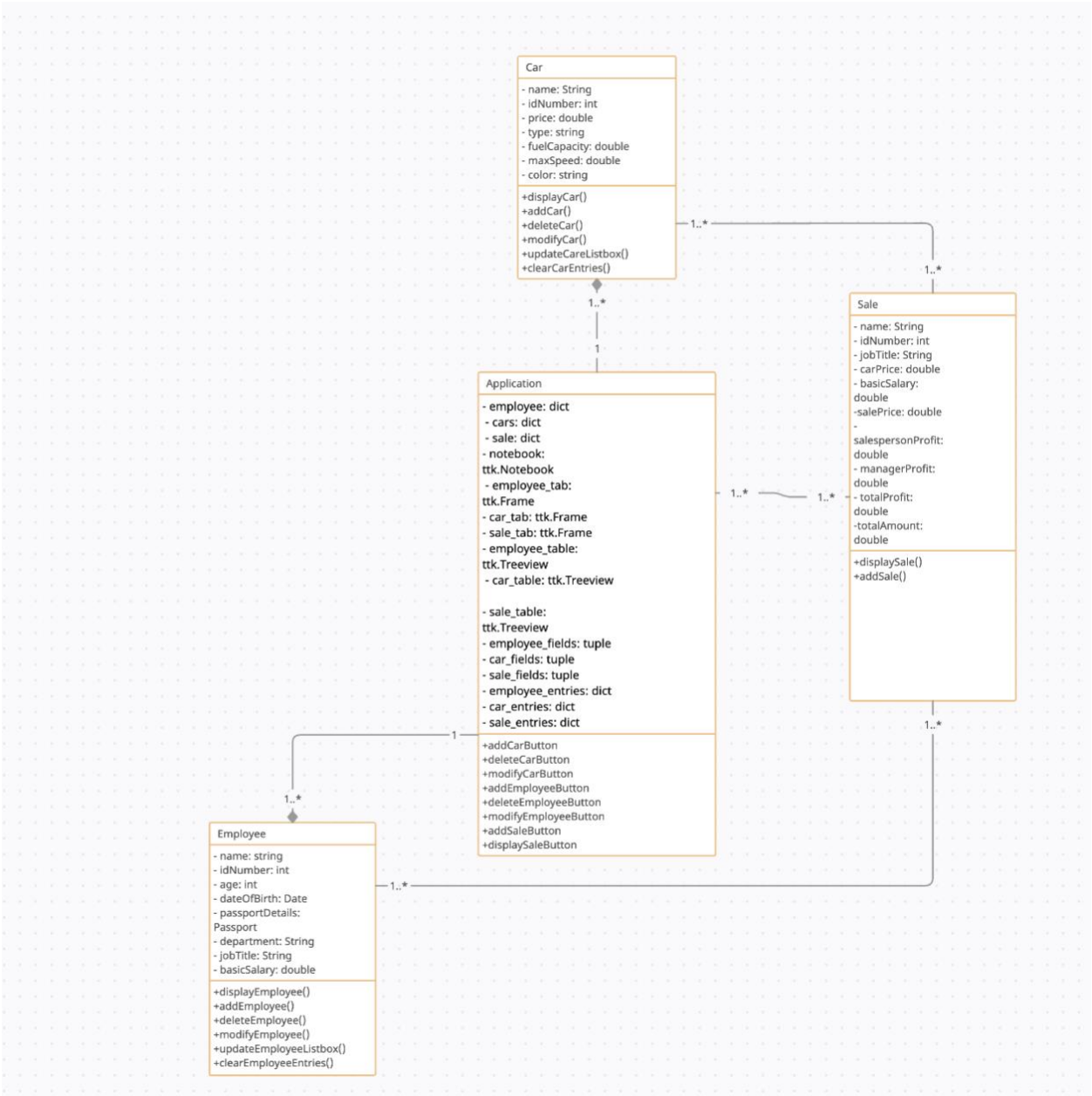
1. #OOAD: Analyze and design software that maps real-world entities and relationships using Unified Modelling Language (UML) notations.
2. #OOProgramming: Create working object-oriented programs in a computer language that are wellstructured, error-free, and can solve computational problems.
3. #SWImplementation: Develop and test interactive multi-tier software applications, that ensures one or more CRUD (create, read, update, and delete) operations.
4. #SWDocumentation: Communicate with a clear and precise style that is suited to an appropriate audience to produce well-documented code, design documents, and presentations that are readable and understandable.

Name	Car Sold	Sale Price
Joy Rogers	ZX3	155000
Joy Rogers	VX3	57800
Joy Rogers	VX3	55000
Joy Rogers	SX3	89000

Joy Rogers	SX3	93000

Mark Jones	VX3	58000
Mark Jones	VX3	58000
Mark Jones	VX3	158000
Mark Jones	VX3	158000
Mark Jones	VX3	158000

A UML class diagram:



Relationship and assumptions:

Employee Attributes:

- Name
- ID Number
- Age
- Date of Birth
- Passport Details
- Department

- Job Title
- Basic Salary

Car Attributes:

- Name
- ID Number
- Price
- Type (hatchback, sedan, SUV)
- Fuel Capacity
- Maximum Speed
- Color

Sale attribute:

- Name
- ID Number
- Job Title
- Car Price
- Basic Salary
- Sale Price
- Salesperson Profit
- Manager Profit
- Total Profit
- Total Amount

In addition to these attributes, we can also include other attributes based on the specific needs of the company. Functionalities:

- Add Employee/Cars/Sale
- Delete Employee/Cars
- Modify Employee/Cars
- Display Sale
- Assign Manager to Salespersons
- Record Sales made by Salespersons
- Calculate Profit for each Sale
- Generate Reports based on Sales, Employees, Cars, etc.

1. Composition relationship between Application and Employee:

This relationship indicates that an Application is composed of one or more Employees.
The cardinality is 1 to many, meaning that one Application can have multiple Employees, but each Employee can belong to only one Application.
The assumption here is that an Employee is part of an Application and cannot exist without being associated with one.

2. Composition relationship between Application and Car:

- This relationship indicates that an Application is composed of one or more Cars.

- The cardinality is 1 to many, meaning that one Application can have multiple Cars, but each Car can belong to only one Application.
- The assumption here is that a Car is part of an Application and cannot exist without being associated with one.

3. Association relationship between Application and Sale:

- This relationship indicates that an Application can be associated with multiple Sales, and a Sale can be associated with multiple Applications.
- The cardinality is many to many, meaning that one Application can have multiple Sales, and one Sale can involve multiple Applications.
- The assumption here is that a Sale can be made through an Application, but a Sale can also exist without being associated with any Application.

4. Association relationship between Sale and Employee:

- This relationship indicates that a Sale can involve multiple Employees, and an Employee can be involved in multiple Sales.
- The cardinality is many to many, meaning that one Sale can involve multiple Employees, and one Employee can be involved in multiple Sales.
- The assumption here is that an Employee can participate in the sale of one or more Cars, but a Sale can also involve multiple Employees.

5. Association relationship between Sale and Car:

- This relationship indicates that a Sale can involve multiple Cars, and a Car can be involved in multiple Sales.
- The cardinality is many to many, meaning that one Sale can involve multiple Cars, and one Car can be involved in multiple Sales.
- The assumption here is that a Sale can involve one or more Cars, but a Car can also be sold without being associated with any Sale.

6. Composition relationship between Application and Employee: This relationship represents the fact that an Application is composed of one or more Employees. The assumption here is that an Application cannot exist without at least one Employee. Furthermore, when an Application is deleted, all associated Employees are also deleted. This relationship is one-to-many, as one Application can have multiple Employees, but each Employee can only belong to one Application.
7. Composition relationship between Application and Car: This relationship represents the fact that an Application is composed of one or more Cars. The assumption here is that an Application cannot exist without at least one Car. Furthermore, when an Application is deleted, all associated Cars are also deleted. This relationship is one-to-many, as one Application can have multiple Cars, but each Car can only belong to one Application.
8. Association relationship between Application and Sale: This relationship represents the fact that an Application can be associated with multiple Sales, and each Sale can be

associated with multiple Applications. The assumption here is that an Application may have one or more Sales associated with it, and that a Sale may involve one or more Applications. This relationship is many-to-many.

9. Association relationship between Sale and Employee: This relationship represents the fact that a Sale can involve multiple Employees, and each Employee can be involved in multiple Sales. The assumption here is that a Sale may require the participation of multiple Employees, and that an Employee may participate in multiple Sales. This relationship is many-to-many.
10. Association relationship between Sale and Car: This relationship represents the fact that a Sale can involve multiple Cars, and each Car can be involved in multiple Sales. The assumption here is that a Sale may involve the sale of multiple Cars, and that a Car may be involved in multiple Sales. This relationship is many-to-many.

Employee Class: The Employee class represents an employee working in the company. It has attributes such as Name, ID Number, Age, Date of Birth, Passport Details, Department, Job Title, and Basic Salary. The Name attribute represents the full name of the employee. The ID Number attribute is a unique identifier for the employee. The Age and Date of Birth attributes represent the employee's age and date of birth. The Passport Details attribute contains information about the employee's passport. The Department attribute represents the department the employee belongs to. The Job Title attribute represents the employee's job title or position in the company. The Basic Salary attribute represents the employee's basic salary.
Car Class: The Car class represents a car sold by the company. It has attributes such as Name, ID Number, Price, Type, Fuel Capacity, Maximum Speed, and Color. The Name attribute represents the car's name. The ID Number attribute is a unique identifier for the car. The Price attribute represents the price of the car. The Type attribute represents the type of car, such as hatchback, sedan, or SUV. The Fuel Capacity attribute represents the maximum amount of fuel the car can hold. The Maximum Speed attribute represents the car's maximum speed. The Color attribute represents the color of the car.

Sale Class: The Sale class represents a sale made by the company. It has attributes such as Name, ID Number, Job Title, Car Price, Basic Salary, Sale Price, Salesperson Profit, Manager Profit, Total Profit, and Total Amount. The Name attribute represents the name of the sale. The ID Number attribute is a unique identifier for the sale. The Job Title attribute represents the job title or position of the person who made the sale. The Car Price attribute represents the price of the car sold in the sale. The Basic Salary attribute represents the basic salary of the salesperson. The Sale Price attribute represents the price at which the car was sold in the sale. The Salesperson Profit attribute represents the profit made by the salesperson from the sale. The Manager Profit attribute represents the profit made by the manager from the sale. The Total Profit attribute represents the total profit made by the company from the sale. The Total Amount attribute represents the total amount of money made from the sale.

The Employee, Car, and Sale classes have different attributes, but they are all related to the company's operations. The Employee class represents the employees who work for the company, the Car class represents the cars sold by the company, and the Sale class represents the sales made by the company. The attributes of each class provide information about the objects they represent and their relationships with other objects in the system.

The functionalities listed in the description of the system provide different ways in which the classes and their attributes can be used to perform operations and manage the system. The functionalities allow the user to add, delete, and modify objects in the system, display information, assign managers to salespersons, record sales, calculate profit, and generate reports. These functionalities provide a comprehensive way of managing the system and ensuring that it operates efficiently.

The python code documented:

```
import tkinter as tk #importing the tkinter library for GUI application
from tkinter import ttk # importing tkinter themed widgets to provide more styling options
from tkinter import messagebox # importing tkinter message box to show messages to the user
import pickle # importing pickle library to store data in binary files

# employee details
employees = [
    {"Name": "Susan Meyers", "ID": 47899, "Department": "Accounting", "Job Title": "Manager", "Basic Salary": 37500},
    {"Name": "Mark Jones", "ID": 39119, "Department": "IT", "Job Title": "Salesperson", "Basic Salary": 26000},
    {"Name": "Joy Rogers", "ID": 81774, "Department": "Manufacturing", "Job Title": "Salesperson", "Basic Salary": 24000}
]

# car details
cars = [
    {"Name": "Jazz VX3", "ID": "55,000", "Number": "Hatch", "Price": 55000},
    {"Name": "Mark3 SX3", "ID": "84,000", "Number": "Sedan", "Price": 84000},
    {"Name": "Wagoner ZX3", "ID": "125,000", "Number": "SUV", "Price": 125000}
]

# storing employee details in binary file
with open('employees.pickle', 'wb') as f:
    pickle.dump(employees, f)

# storing car details in binary file
with open('cars.pickle', 'wb') as f:
    pickle.dump(cars, f)


# employee details
employees = [
    {"Name": "Susan Meyers", "ID": 47899, "Department": "Accounting", "Job Title": "Manager", "Basic Salary": 37500},
    {"Name": "Mark Jones", "ID": 39119, "Department": "IT", "Job Title": "Salesperson", "Basic Salary": 26000},
```

```
    {"Name": "Joy Rogers", "ID": 81774, "Department": "Manufacturing", "Job Title":
"Salesperson", "Basic Salary": 24000}
]

# car details
cars = [
    {"Name": "Jazz VX3", "ID": "55,000", "Number": "Hatch", "Price": 55000},
    {"Name": "Mark3 SX3", "ID": "84,000", "Number": "Sedan", "Price": 84000},
    {"Name": "Wagoner ZX3", "ID": "125,000", "Number": "SUV", "Price": 125000}
]

# storing employee details in binary file
with open('employees.pickle', 'wb') as f:
    pickle.dump(employees, f)

# storing car details in binary file
with open('cars.pickle', 'wb') as f:
    pickle.dump(cars, f)
```

explanation of the code above: The code is mainly responsible for importing required libraries, storing the employee and car details in a list, and then storing the details in binary files using pickle library. This code is not creating any user interface. The first section imports the libraries required to create the user interface. The second section initializes the employee and car details. The third and fourth sections store the employee and car details in a binary file. Finally, the fifth section repeats the second and third sections of the code, which is redundant. The code can be optimized by removing the fifth section.

```
class Employee: # defining a class called "Employee"
    # initializing an instance of the class with attributes of the employee
    def __init__(self, name, id_number, age, dob, passport, department, job_title, basic_salary):
        self.name = name
        self.id_number = id_number
        self.age = age
        self.dob = dob
        self.passport = passport
        self.department = department
        self.job_title = job_title
        self.basic_salary = basic_salary

class Car: # defining a class called "Car"
    # initializing an instance of the class with attributes of the car
    def __init__(self, name, id_number, price, car_type, fuel_capacity, maximum_speed, color):
        self.name = name
        self.id_number = id_number
        self.price = price
        self.car_type = car_type
        self.fuel_capacity = fuel_capacity
```

```
        self.maximum_speed = maximum_speed
        self.color = color

class Sale: # defining a class called "Sale"
    # initializing an instance of the class with attributes of the sale
    def __init__(self, name, id_number, job_title, car_price, basic_salary):
        self.name = name
        self.id_number = id_number
        self.job_title = job_title
        self.car_price = car_price
        self.basic_salary = basic_salary
        # if the job title is Manager, set profit percentage to 0.035, otherwise if Sales Person, set it
to 0.065
        if self.job_title == "Manager":
            self.profit_percentage = 0.035
        elif self.job_title == "Sales Person":
            self.profit_percentage = 0.065
    # calculating the profit based on the sale price and returning the salesperson profit, manager
profit, total profit, and total amount earned
    def calculate_profit(self, sale_price):
        profit = sale_price - self.car_price
        salesperson_profit = profit * 0.065
        manager_profit = profit * 0.035
        total_profit = salesperson_profit + manager_profit
        total_amount = self.basic_salary + total_profit
        return (salesperson_profit, manager_profit, total_profit, total_amount)


class Application(tk.Tk): #Create a class named Application that inherits from the Tk class
    def __init__(self): # Constructor method for the Application class
        # Initialize empty dictionaries for employee, cars, and sales
        self.employee = {}
        self.cars = {}
        self.sale = {}

        super().__init__() # Call the constructor of the Tk class using the super() function
        self.title("B&M Car Sales") # Set the title of the main window
        self.geometry("600x700")  # Set the size of the main window
        self.resizable(False, False) # Disable the resizing of the main window

        # Create a notebook widget and attach it to the main window
        self.notebook = ttk.Notebook(self)
        self.notebook.pack(fill="both", expand=True)

        # Create a tab for employee details and add it to the notebook
        self.employee_tab = ttk.Frame(self.notebook)
        self.notebook.add(self.employee_tab, text="Employees")

        # Create a tab for car details and add it to the notebook
```

```
        self.car_tab = ttk.Frame(self.notebook)
        self.notebook.add(self.car_tab, text="Cars")

        # Create a tab for sales details and add it to the notebook
        self.sale_tab = ttk.Frame(self.notebook)
        self.notebook.add(self.sale_tab, text="Sales")

        # Create a treeview widget for employee details and attach it to the employee tab
        self.employee_table = ttk.Treeview(self.employee_tab)
        self.employee_table.pack(fill="both", expand=True)

        # Create a treeview widget for car details and attach it to the car tab
        self.car_table = ttk.Treeview(self.car_tab)
        self.car_table.pack(fill="both", expand=True)

        # Create a treeview widget for sales details and attach it to the sales tab
        self.sale_table = ttk.Treeview(self.sale_tab)
        self.sale_table.pack(fill="both", expand=True)

        # Create a scrollbar for the car table and attach it to the car tab
        car_table_scroll = ttk.Scrollbar(self.car_tab, orient=tk.VERTICAL,
command=self.car_table.yview)
        car_table_scroll.pack(side=tk.RIGHT, fill=tk.Y)
        self.car_table.configure(yscrollcommand=car_table_scroll.set)

        # Create a scrollbar for the employee table and attach it to the employee tab
        employee_table_scroll = ttk.Scrollbar(self.employee_tab, orient=tk.VERTICAL,
command=self.employee_table.yview)
        employee_table_scroll.pack(side=tk.RIGHT, fill=tk.Y)
        self.employee_table.configure(yscrollcommand=employee_table_scroll.set)

        # Create a scrollbar for the sale table and attach it to the sale tab
        sale_table_scroll = ttk.Scrollbar(self.sale_tab, orient=tk.VERTICAL,
command=self.sale_table.yview)
        sale_table_scroll.pack(side=tk.RIGHT, fill=tk.Y)
        self.sale_table.configure(yscrollcommand=sale_table_scroll.set)

        # Define employee fields
        self.employee_fields = ("Name", "ID Number", "Age", "DOB", "Passport", "Department",
"Job Title", "Basic Salary") #Set employee_fields as a tuple of employee attributes
        self.employee_table["columns"] = self.employee_fields # Set columns of employee_table as
employee_fields
        for field in self.employee_fields:    # Add headings for each column of employee_table
            self.employee_table.heading(field, text=field)     # Set the width and stretch of column 0
of employee_table
        self.employee_table.column("#0", width=0, stretch=tk.NO)    # Set the width and stretch of
all other columns of employee_table
        # Set the width and stretch of all other columns of employee_table
        for col in self.employee_table["columns"]:
```

```
            self.employee_table.column(col, width=100, stretch=tk.NO)


        self.car_fields = ("Name", "ID Number", "Price", "Type", "Fuel Capacity", "Maximum Speed", "Color")    # Set car_fields as a tuple of car attributes
        self.car_table["columns"] = self.car_fields # Set columns of car_table as car_fields
        # Add headings for each column of car_table
        for field in self.car_fields:
            self.car_table.heading(field, text=field)
        self.car_table.column("#0", width=0, stretch=tk.NO)    # Set the width and stretch of column 0 of car_table
        # Set the width and stretch of all other columns of car_table
        for col in self.car_table["columns"]:
            self.car_table.column(col, width=100, stretch=tk.NO)

        self.sale_fields = ("Name", "ID Number", "Job Title", "Car Price", "Basic Salary", "Sale Price", "Salesperson Profit", "Manager Profit", "Total Profit", "Total Amount")    # Set sale_fields as a tuple of sale attributes
        self.sale_table["columns"] = self.sale_fields    # Set columns of sale_table as sale_fields
        # Add headings for each column of sale_table
        for field in self.sale_fields:
            self.sale_table.heading(field, text=field)
        self.sale_table.column("#0", width=0, stretch=tk.NO)    # Set the width and stretch of column 0 of sale_table
        # Set the width and stretch of all other columns of sale_table
        for col in self.sale_table["columns"]:
            self.sale_table.column(col, width=100, stretch=tk.NO)


        self.employee_entries = {}    # Create an empty dictionary employee_entries to store employee attributes
        # Create a frame for each employee attribute and pack them in the employee_tab
        # Store each entry widget in employee_entries dictionary
        for field in self.employee_fields:
            frame = ttk.Frame(self.employee_tab)
            frame.pack(side=tk.TOP, fill=tk.X)
            label = ttk.Label(frame, text=field, width=15)
            label.pack(side=tk.LEFT, padx=5, pady=5)
            entry = ttk.Entry(frame, width=30)
            entry.pack(side=tk.LEFT, padx=5, pady=5)
            self.employee_entries[field] = entry


        self.car_entries = {}    # Create an empty dictionary car_entries to store car attributes
        # Create a frame for each car attribute and pack them in the car_tab
        # Store each entry widget in car_entries dictionary
        for field in self.car_fields:
            frame = ttk.Frame(self.car_tab)
            frame.pack(side=tk.TOP, fill=tk.X)
```

```
            label = ttk.Label(frame, text=field, width=15)
            label.pack(side=tk.LEFT, padx=5, pady=5)
            entry = ttk.Entry(frame, width=30)
            entry.pack(side=tk.LEFT, padx=5, pady=5)
            self.car_entries[field] = entry


        self.sale_entries = {} # Creates an empty dictionary to store the sale entries
        for field in self.sale_fields: # Loop through each field in the sale fields
            if field not in ["Sale Price", "Salesperson Profit", "Manager Profit", "Total Profit", "Total
Amount"]:     # If the field is not a computed field
                # Create a frame to hold the label and entry widgets
                frame = ttk.Frame(self.sale_tab)
                frame.pack(side=tk.TOP, fill=tk.X)
                # Create a label with the name of the field
                label = ttk.Label(frame, text=field, width=15)
                label.pack(side=tk.LEFT, padx=5, pady=5)
                # Create an entry widget to get input for the field
                entry = ttk.Entry(frame, width=30)
                entry.pack(side=tk.LEFT, padx=5, pady=5)
                self.sale_entries[field] = entry # Store the entry widget in the sale_entries dictionary


        # Buttons to manage car details
        # Create buttons to manage car details
        car_button_frame = ttk.Frame(self.car_tab)
        car_button_frame.pack(side=tk.BOTTOM, fill=tk.X)

        # Create a button to add a car
        add_car_button = ttk.Button(car_button_frame, text="Add Car", command=self.add_car)
        add_car_button.pack(side=tk.LEFT, padx=5, pady=5)

        # Create a button to delete a car
        delete_car_button = ttk.Button(car_button_frame, text="Delete Car",
command=self.delete_car)
        delete_car_button.pack(side=tk.LEFT, padx=5, pady=5)

        # Create a button to modify a car
        modify_car_button = ttk.Button(car_button_frame, text="Modify Car",
command=self.modify_car)
        modify_car_button.pack(side=tk.LEFT, padx=5, pady=5)

        # Create a frame to hold the employee buttons
        employee_button_frame = ttk.Frame(self.employee_tab)
        employee_button_frame.pack(side=tk.BOTTOM, fill=tk.X)

        # Create a button to add an employee
        add_employee_button = ttk.Button(employee_button_frame, text="Add Employee",
command=self.add_employee)
```

```
        add_employee_button.pack(side=tk.LEFT, padx=5, pady=5)

        # Create a button to delete an employee
        delete_employee_button = ttk.Button(employee_button_frame, text="Delete Employee",
command=self.delete_employee)
        delete_employee_button.pack(side=tk.LEFT, padx=5, pady=5)

        # Create a button to modify an employee
        modify_employee_button = ttk.Button(employee_button_frame, text="Modify Employee",
command=self.modify_employee)
        modify_employee_button.pack(side=tk.LEFT, padx=5, pady=5)

        # Create a frame to hold the sale buttons
        sale_button_frame = ttk.Frame(self.sale_tab)
        sale_button_frame.pack(side=tk.BOTTOM, fill=tk.X)

        # Create a button to add a sale
        add_sale_button = ttk.Button(sale_button_frame, text="Add Sale",
command=self.add_sale)
        add_sale_button.pack(side=tk.LEFT, padx=5, pady=5)

        # Create a button to display the sales
        sale_button = ttk.Button(sale_button_frame, text="Display Sale",
command=self.display_sale)
        sale_button.pack(side=tk.LEFT, padx=5, pady=5)


    def display_employee(self): # This is a method called "display_employee" that takes in a
"self" parameter.
        # Get employee ID number from the entry field
        id_number = self.employee_entries['ID Number'].get()

        # Check if ID number is valid
        if not id_number:
            messagebox.showerror("Error", "Please enter an ID number.")        # Show an error
message if the ID number is not entered
            return
        try:
            id_number = int(id_number)        # Convert the ID number to an integer
        except ValueError:
            messagebox.showerror("Error", "Invalid ID number.")        # Show an error message if
the ID number is not a valid integer
            return
        if id_number not in self.employee:
            messagebox.showerror("Error", "Employee data not found.")        # Show an error
message if the ID number is not found in the employee data
            return

        # Display employee details in the entry fields
```

```
        # Display employee details in the entry fields
        employee = self.employee[id_number]
        # Delete and insert employee name into the name entry field
        self.employee_entries['Name'].delete(0, tk.END)
        self.employee_entries['Name'].insert(0, employee.name)
        # Delete and insert employee age into the age entry field
        self.employee_entries['Age'].delete(0, tk.END)
        self.employee_entries['Age'].insert(0, employee.age)
        # Delete and insert employee date of birth into the DOB entry field
        self.employee_entries['DOB'].delete(0, tk.END)
        self.employee_entries['DOB'].insert(0, employee.dob)
        # Delete and insert employee passport number into the passport entry field
        self.employee_entries['Passport'].delete(0, tk.END)
        self.employee_entries['Passport'].insert(0, employee.passport)
        # Delete and insert employee department into the department entry field
        self.employee_entries['Department'].delete(0, tk.END)
        self.employee_entries['Department'].insert(0, employee.department)
        # Delete and insert employee job title into the job title entry field
        self.employee_entries['Job Title'].delete(0, tk.END)
        self.employee_entries['Job Title'].insert(0, employee.job_title)
        # Delete and insert employee basic salary into the basic salary entry field
        self.employee_entries['Basic Salary'].delete(0, tk.END)
        self.employee_entries['Basic Salary'].insert(0, employee.basic_salary)
        # Insert employee ID number into the ID Number entry field
        self.employee_entries['ID Number'].delete(0, tk.END)
        self.employee_entries['ID Number'].insert(0, id_number)

    def display_car(self): # This is a method called "display_car" that takes in a "self" parameter.
        # Get car ID number from the entry field
        id_number = self.car_entries['ID Number'].get() #This line gets the ID number of the car
that the user wants to display from the "ID Number" entry field.

        # Check if ID number is valid
        # This line checks if the ID number is empty, and if it is, it displays an error message and
returns.
        if not id_number:
            messagebox.showerror("Error", "Please enter an ID number.")        # Show an error
message if the ID number is empty
            return
        #This block of code tries to convert the ID number to an integer, and if it can't be converted,
it displays an error message and returns.
        try:
            id_number = int(id_number)
        except ValueError:
            messagebox.showerror("Error", "Invalid ID number.")        # Show an error message if
the ID number is not a valid integer
            return
        #This line checks if the ID number is in the dictionary of cars. If it isn't, it displays an error
message and returns.
```

```
        if id_number not in self.cars:
            messagebox.showerror("Error", "Car data not found.")        # Show an error message if
the car with the given ID number does not exist
            return

        # Display car details in the entry fields just like employee
        car = self.cars[id_number]
        # Clear the Name entry field and insert the name of the car
        self.car_entries['Name'].delete(0, tk.END)
        self.car_entries['Name'].insert(0, car.name)
        # Clear the Price entry field and insert the price of the car
        self.car_entries['Price'].delete(0, tk.END)
        self.car_entries['Price'].insert(0, car.price)
        # Clear the Type entry field and insert the type of the car
        self.car_entries['Type'].delete(0, tk.END)
        self.car_entries['Type'].insert(0, car.car_type)
        # Clear the Fuel Capacity entry field and insert the fuel capacity of the car
        self.car_entries['Fuel Capacity'].delete(0, tk.END)
        self.car_entries['Fuel Capacity'].insert(0, car.fuel_capacity)
        # Clear the Maximum Speed entry field and insert the maximum speed of the car
        self.car_entries['Maximum Speed'].delete(0, tk.END)
        self.car_entries['Maximum Speed'].insert(0, car.maximum_speed)
        # Clear the Color entry field and insert the color of the car
        self.car_entries['Color'].delete(0, tk.END)
        self.car_entries['Color'].insert(0, car.color)
        # Insert car ID number into the ID Number entry field
        self.car_entries['ID Number'].delete(0, tk.END)
        self.car_entries['ID Number'].insert(0, id_number)


    def display_sale(self):
        # Get sale ID number from the entry field
        id_number = self.sale_entries['ID Number'].get()

        # Check if ID number is valid
        if not id_number:
            messagebox.showerror("Error", "Please enter an ID number.")        # Display error
message if ID number is empty
            return
        try:
            # Try to convert ID number to integer
            id_number = int(id_number)
        except ValueError:
            messagebox.showerror("Error", "Invalid ID number.")        # Display error message if ID
number is not a valid integer
            return
        if id_number not in self.sale:
            messagebox.showerror("Error", "Sale data not found.")        # Display error message if
sale data with given ID number is not found
```

```
            return

        # Display sale details in the table
        sale = self.sale[id_number]
        self.sale_table.item(id_number, values=(
            sale.name,
            sale.id_number,
            sale.job_title,
            sale.car_type,
            sale.car_price,
            sale.basic_salary,
            '',  # salesperson profit column
            '',  # manager profit column
            '',  # total profit column
        ))

        # Calculate salesperson profit, manager profit, and total profit
        if sale.job_title == 'Sales Person':
            # Calculate salesperson profit if the sale was made by a salesperson
            salesperson_profit = 0.065 * sale.car_price
            manager_profit = 0
        elif sale.job_title == 'Manager':
            # Calculate salesperson and manager profits if the sale was made by a manager
            salesperson_profit = 0.035 * sale.car_price
            manager_profit = 0.01 * sale.car_price
        total_profit = salesperson_profit + manager_profit

        # Update salesperson profit, manager profit, and total profit columns in the table
        self.sale_table.set(id_number, column='Salesperson Profit', value=salesperson_profit)
        self.sale_table.set(id_number, column='Manager Profit', value=manager_profit)
        self.sale_table.set(id_number, column='Total Profit', value=total_profit)


    def add_employee(self):
        # Get employee information from the entry fields
        name = self.employee_entries['Name'].get()
        id_number_str = self.employee_entries['ID Number'].get()
        age = self.employee_entries['Age'].get()
        dob = self.employee_entries['DOB'].get()
        passport = self.employee_entries['Passport'].get()
        department = self.employee_entries['Department'].get()
        job_title = self.employee_entries['Job Title'].get()
        basic_salary = self.employee_entries['Basic Salary'].get()

        # Check if ID number is valid
        if not id_number_str:
            messagebox.showerror("Error", "Please enter an ID number.")
            return
        try:
```

```
            id_number = int(id_number_str)
        except ValueError:
            messagebox.showerror("Error", "Invalid ID number. Please enter an integer value.")
            return
        if id_number in self.employee:
            messagebox.showerror("Error", "ID number already exists.")
            return

        # Create a new employee object and add it to the employees dictionary
        employee = Employee(name, id_number, age, dob, passport, department, job_title,
basic_salary)
        self.employee[id_number] = employee

        # Insert the new employee into the employee table
        self.employee_table.insert("", "end", iid=id_number, text=id_number, values=(name,
id_number, age, dob, passport, department, job_title, basic_salary))



        # Clear the entry fields
        for entry in self.employee_entries.values():
            entry.delete(0, "end")

        # Display a success message
        messagebox.showinfo("Success", "Employee added successfully.")


    def update_employee_listbox(self):
        # Delete all current entries in the employee table
        self.employee_table.delete(*self.employee_table.get_children())

        # Insert all employees in the employees dictionary into the employee table
        for id_number, employee in self.employee.items():
            self.employee_table.insert("", "end", iid=id_number, text=id_number, values=(name,
id_number, age, dob, passport, department, job_title, basic_salary))

    def clear_employee_entries(self):# Defines a method called 'clear_employee_entries' which
takes self as the argument.
        for entry in self.employee_entries.values():    # Iterates through the values of the dictionary
'employee_entries' and assigns each value to the variable 'entry'.
            entry.delete(0, "end")        # Deletes the characters from index 0 to the end of the entry
widget.

    def delete_employee(self):
        # Get the selected employee id number
        selected_item = self.employee_table.selection()
        if not selected_item:
            messagebox.showerror("Error", "No employee selected!")
            return
        selected_id_number = self.employee_table.item(selected_item)['values'][1]
```

```
    # Check if the employee exists in the database
    if selected_id_number not in self.employee:
        messagebox.showerror("Error", f"Employee with ID {selected_id_number} not found!")
        return

    # Remove employee from the database
    del self.employee[selected_id_number]

    # Update the employee listbox
    self.update_employee_listbox()

    # Clear the entry fields
    self.clear_employee_entries()

    # Display a success message
    messagebox.showinfo("Success", "Employee deleted successfully.")

def modify_employee(self):
    # Get the selected employee id number
    selected_item = self.employee_table.selection()
    if not selected_item:
        messagebox.showerror("Error", "No employee selected!")
        return
    selected_id_number = self.employee_table.item(selected_item)['values'][1]

    # Check if the employee exists in the database
    if selected_id_number not in self.employee:
        messagebox.showerror("Error", f"Employee with ID {selected_id_number} not found!")
        return

    # Get employee data from the entry fields
    name = self.employee_entries['Name'].get()
    age = self.employee_entries['Age'].get()
    dob = self.employee_entries['DOB'].get()
    passport = self.employee_entries['Passport'].get()
    department = self.employee_entries['Department'].get()
    job_title = self.employee_entries['Job Title'].get()
    basic_salary = self.employee_entries['Basic Salary'].get()

    # Check if any field is left empty
    if not (name and department and job_title and basic_salary):
        messagebox.showerror("Error", "All fields are required!")
        return

    # Modify the employee data in the database
    self.employee[selected_id_number].name = name
    self.employee[selected_id_number].age = age
    self.employee[selected_id_number].dob = dob
```

```
        self.employee[selected_id_number].passport = passport
        self.employee[selected_id_number].department = department
        self.employee[selected_id_number].job_title = job_title
        self.employee[selected_id_number].basic_salary = basic_salary

        # Update the employee table
        self.employee_table.item(selected_item, text=selected_id_number, values=(name,
selected_id_number, age, dob, passport, department, job_title, basic_salary))

        # Clear the entry fields
        self.clear_employee_entries()

        # Display a success message
        messagebox.showinfo("Success", "Employee modified successfully.")

    def add_car(self):
        # Get car data from the entry fields
        name = self.car_entries['Name'].get()
        id_number_str = self.car_entries['ID Number'].get()
        price = self.car_entries['Price'].get()
        car_type = self.car_entries['Type'].get()
        fuel_capacity = self.car_entries['Fuel Capacity'].get()
        maximum_speed = self.car_entries['Maximum Speed'].get()
        color = self.car_entries['Color'].get()

        # Check if ID number is valid
        if not id_number_str:
            messagebox.showerror("Error", "Please enter an ID number.")
            return
        try:
            id_number = int(id_number_str)
        except ValueError:
            messagebox.showerror("Error", "Invalid ID number. Please enter an integer value.")
            return
        if id_number in self.cars:
            messagebox.showerror("Error", "ID number already exists.")
            return

        # Create a new car object and add it to the cars dictionary
        car = Car(name, id_number, price, car_type, fuel_capacity, maximum_speed, color)
        self.cars[id_number] = car

        # Insert the new car into the car table
        self.car_table.insert("", "end", iid=id_number, text=id_number, values=(name, id_number,
price, car_type, fuel_capacity, maximum_speed, color))

        # Clear the entry fields
        for entry in self.car_entries.values():
            entry.delete(0, "end")
```

```
        # Display a success message
        messagebox.showinfo("Success", "Car added successfully.")

    def update_car_listbox(self):
        # Delete all current entries in the car table
        self.car_table.delete(*self.car_table.get_children())

        # Insert all cars in the cars dictionary into the car table
        for id_number, car in self.cars.items():
            self.car_table.insert("", "end", iid=id_number, text=id_number, values=(car.name,
id_number, car.price, car.car_type, car.fuel_capacity, car.maximum_speed, car.color))

    def clear_car_entries(self): # This is a method named 'clear_car_entries' within a class
        for entry in self.car_entries.values():     # This is a loop iterating through values of a
dictionary
            entry.delete(0, "end")       # This is a method to delete text in a tkinter entry widget from
start index 0 to the end index.


    def delete_car(self):
        # Get the selected car id number
        selected_item = self.car_table.selection()
        if not selected_item:
            messagebox.showerror("Error", "No car selected!")
            return
        selected_id_number = self.car_table.item(selected_item)['values'][1]

        # Check if the car exists in the database
        if selected_id_number not in self.cars:
            messagebox.showerror("Error", f"Car with ID {selected_id_number} not found!")
            return

        # Remove car from the database
        del self.cars[selected_id_number]

        # Update the car listbox
        self.update_car_listbox()

        # Clear the entry fields
        self.clear_car_entries()

        # Display a success message
        messagebox.showinfo("Success", "Car deleted successfully.")

    def modify_car(self):
        # Get the selected car id number
        selected_item = self.car_table.selection()
        if not selected_item:
```

```
            messagebox.showerror("Error", "No car selected!")
            return
        selected_id_number = self.car_table.item(selected_item)['values'][1]

        # Check if the car exists in the database
        if selected_id_number not in self.cars:
            messagebox.showerror("Error", f"Car with ID {selected_id_number} not found!")
            return

        # Get car data from the entry fields
        name = self.car_entries['Name'].get()
        price = self.car_entries['Price'].get()
        car_type = self.car_entries['Type'].get()
        fuel_capacity = self.car_entries['Fuel Capacity'].get()
        maximum_speed = self.car_entries['Maximum Speed'].get()
        color = self.car_entries['Color'].get()

        # Check if any field is left empty
        if not (name and price and car_type and fuel_capacity and maximum_speed and color):
            messagebox.showerror("Error", "All fields are required!")
            return

        # Modify the car data in the database
        self.cars[selected_id_number].name = name
        self.cars[selected_id_number].price = price
        self.cars[selected_id_number].car_type = car_type
        self.cars[selected_id_number].fuel_capacity = fuel_capacity
        self.cars[selected_id_number].maximum_speed = maximum_speed
        self.cars[selected_id_number].color = color

        # Update the car table
        self.car_table.item(selected_item, text=selected_id_number, values=(name,
selected_id_number, price, car_type, fuel_capacity, maximum_speed, color))

        # Clear the entry fields
        self.clear_car_entries()

        # Display a success message
        messagebox.showinfo("Success", "Car modified successfully.")

    def add_sale(self):
        # Get data
        name = self.sale_entries["Name"].get()
        id_number = self.sale_entries["ID Number"].get()
        job_title = self.sale_entries["Job Title"].get()
        car_price = float(self.sale_entries["Car Price"].get())
        basic_salary = float(self.sale_entries["Basic Salary"].get())

        # Create Sale object
```

```
        sale = Sale(name, id_number, job_title, car_price, basic_salary)

        # Calculate profits
        salesperson_profit, manager_profit, total_profit, total_amount =
sale.calculate_profit(sale_price)

        # Add sale to dictionary
        self.sale[id_number] = {
            "Name": name,
            "ID Number": id_number,
            "Job Title": job_title,
            "Car Price": car_price,
            "Basic Salary": basic_salary,
            "Sale Price": sale_price,
            "Salesperson Profit": salesperson_profit,
            "Manager Profit": manager_profit,
            "Total Profit": total_profit,
            "Total Amount": total_amount,
        }

        # Clear entry fields
        for entry in self.sale_entries.values():
            entry.delete(0, tk.END)

        # Show success message
        messagebox.showinfo("Sale Added", "Sale successfully added!")


#test case:
if __name__ == "__main__": #The following code checks whether the module is being run as
the main program or not.
    # If the module is being run as the main program, create an instance of the Application class
and start the GUI event loop.
    app = Application()
    app.mainloop()
```

Link to the public GitHub:

https://github.com/fatmanassiri/assignment-3