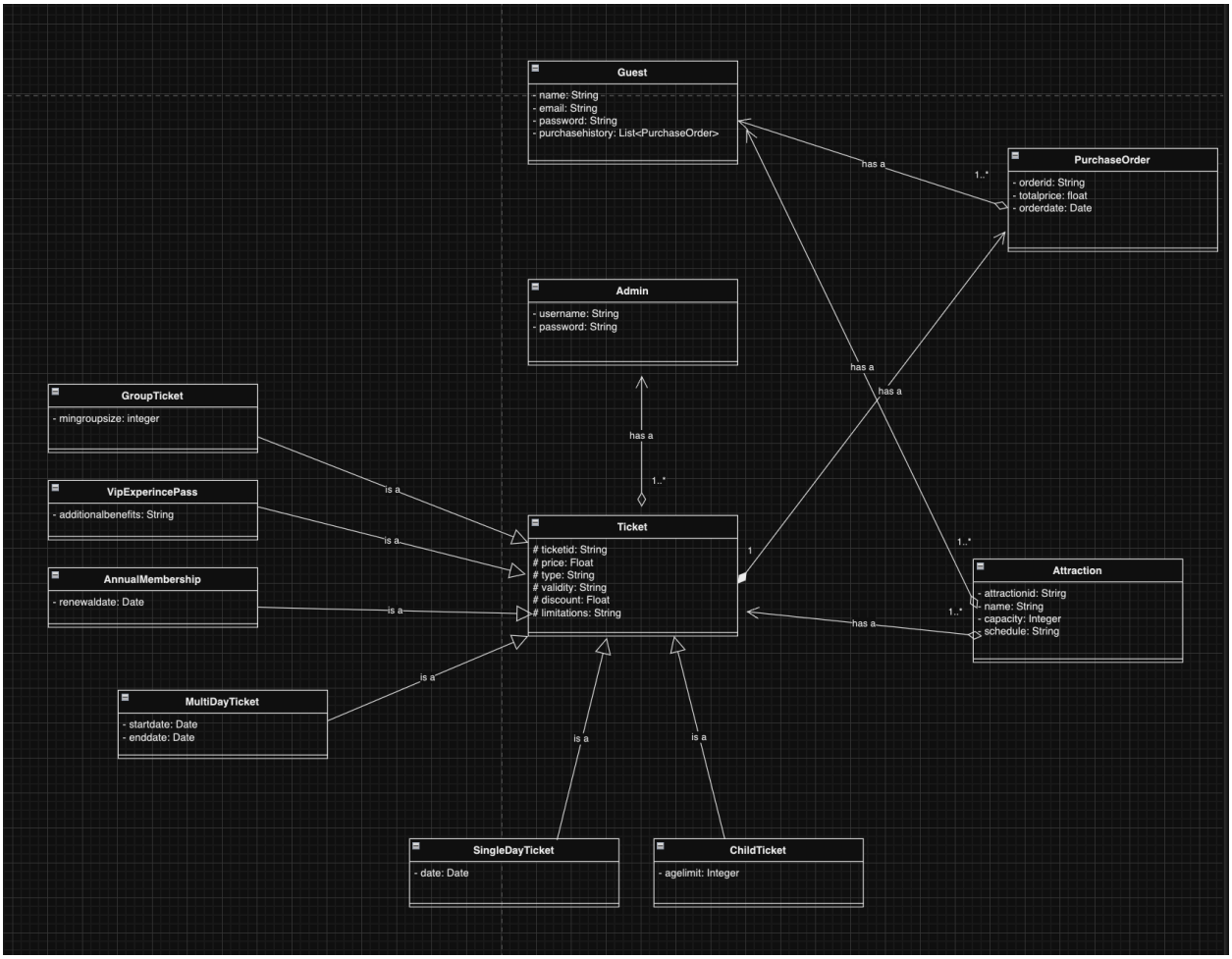
ICS220 - Assignment 3
Mohammad Abdullah Hashim 202107285
Abdulrahman Mohammed baharoon 202234125

- **Design a **UML class diagram **representing the concepts and relationships in the scenario. Ensure the use of 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.**



Modularity

1. Guest:

• Represents a park visitor and manages their personal information (name, email,

password) and their associated purchase orders. Encapsulates the relationship between

guests and attractions they visit.

• Provides methods to track attractions visited and orders made by the guest.

2. Admin:

• Represents a system administrator responsible for managing tickets. This class ensures

the proper configuration and validation of tickets.

3. Ticket:

• Serves as the base class for all ticket types (SingleDayTicket, MultiDayTicket,

AnnualMembership, ChildTicket, GroupTicket, VIPExperiencePass).

• Contains core attributes such as ticket ID, price, validity, and limitations. Each subclass

specializes the ticket type with additional attributes.

4. Attraction:

• Represents park attractions such as rides or shows, with details like capacity and

schedule. Tracks relationships with guests and tickets.

5. PurchaseOrder:

• Manages ticket purchases, representing a composition relationship between tickets and

purchase orders. Stores order ID, total price, and order date.

Relationships

1. **Guest → PurchaseOrder:**
   There is an aggregation relationship between Guest and PurchaseOrder. Each Guest can have multiple PurchaseOrders that track their ticket purchases. The existence of a PurchaseOrder is linked to a Guest but remains independent, allowing the system to manage orders separately from Guest data.

2. **PurchaseOrder → Ticket:**
   There is a composition relationship between PurchaseOrder and Ticket. A PurchaseOrder must include one or more Tickets, and the Tickets cannot exist without a corresponding PurchaseOrder. This ensures that all tickets are associated with a transaction and cannot exist independently.

3. **Ticket → Attraction:**
   There is an aggregation relationship between Ticket and Attraction. A Ticket can grant access to one or more Attractions, but the existence of Attractions is independent of the Ticket. This relationship ensures that tickets reference the attractions they provide access to without creating a dependency, allowing for flexibility in managing attractions separately from the ticketing system.

4. **Guest → Attraction:**
   There is an aggregation relationship between Guest and Attraction. A Guest can visit multiple Attractions during their time at the park, and Attractions can host multiple Guests. The Guest's visit history is linked to Attractions but does not affect the independent existence of the Attractions.

5. **Admin → Ticket:**
   There is an association relationship between Admin and Ticket. Admins are responsible for managing and configuring tickets within the system. This

relationship allows Admins to perform administrative tasks without directly affecting the existence of Tickets, ensuring separation of responsibilities.

6. **Ticket → Subclasses (Inheritance):**
   There is an inheritance relationship between Ticket and its subclasses (e.g., SingleDayTicket, MultiDayTicket, AnnualMembership). Each subclass specializes the Ticket class by adding specific attributes and behaviors, allowing the system to manage different ticket types flexibly.

Assumptions

1. **One PurchaseOrder per Transaction:**
   - Each transaction creates a single PurchaseOrder, containing all tickets purchased.
2. **Multiple Tickets per Attraction:**
   - Tickets can grant access to multiple Attractions, and Attractions can accept multiple types of Tickets.
3. **Guests Visiting Attractions:**
   - Guests are linked to Attractions they visit. It is assumed that a Guest might not visit any Attraction (e.g., if a ticket is purchased but not used).
4. **Admin Responsibilities:**
   - Admins do not directly handle guest data but focus on configuring and managing tickets.
5. **Attraction Capacity is Monitored:**
   - Each Attraction has a defined capacity that is tracked for real-time visitor management.

- **Write **Python code **to implement your UML diagram. Ensure that you define test cases to showcase the program features.**

Classes

# Base class: Ticket
class Ticket:

```
    """Represents a general ticket in the theme park system."""
    def __init__(self, ticket_id, price, ticket_type, validity, discount, limitations):
        self.ticket_id = ticket_id  # Unique ID for the ticket
        self.price = price  # Price of the ticket
        self.ticket_type = ticket_type  # Type of ticket (e.g., Single Day, Multi-Day)
        self.validity = validity  # Validity period or conditions for the ticket
        self.discount = discount  # Discount rate for the ticket
        self.limitations = limitations  # Limitations or restrictions for the ticket
        self.attractions = []  # Many-to-Many relationship with attractions

    # Add an attraction to the ticket
    def add_attraction(self, attraction):
        self.attractions.append(attraction)

# Subclasses of Ticket
class SingleDayTicket(Ticket):
    """Represents a single-day ticket in the theme park system."""
    def __init__(self, ticket_id, price, validity, discount, limitations, date):
        super().__init__(ticket_id, price, "Single Day", validity, discount, limitations)
        self.date = date  # The specific date the ticket is valid for

class MultiDayTicket(Ticket):
    """Represents a multi-day ticket in the theme park system."""
    def __init__(self, ticket_id, price, validity, discount, limitations, start_date, end_date):
        super().__init__(ticket_id, price, "Multi Day", validity, discount, limitations)
        self.start_date = start_date  # Start date of the ticket validity
        self.end_date = end_date  # End date of the ticket validity

class AnnualMembership(Ticket):
    """Represents an annual membership ticket in the theme park system."""
    def __init__(self, ticket_id, price, validity, discount, limitations, renewal_date):
        super().__init__(ticket_id, price, "Annual Membership", validity, discount,
limitations)
        self.renewal_date = renewal_date  # Renewal date for the membership

class ChildTicket(Ticket):
    """Represents a child ticket in the theme park system."""
    def __init__(self, ticket_id, price, validity, discount, limitations, age_limit):
        super().__init__(ticket_id, price, "Child", validity, discount, limitations)
        self.age_limit = age_limit  # Age limit for the child ticket
```

```
class GroupTicket(Ticket):
    """Represents a group ticket in the theme park system."""
    def __init__(self, ticket_id, price, validity, discount, limitations, min_group_size):
        super().__init__(ticket_id, price, "Group", validity, discount, limitations)
        self.min_group_size = min_group_size  # Minimum group size required for this
ticket

class VIPExperiencePass(Ticket):
    """Represents a VIP experience ticket in the theme park system."""
    def __init__(self, ticket_id, price, validity, discount, limitations, additional_benefits):
        super().__init__(ticket_id, price, "VIP", validity, discount, limitations)
        self.additional_benefits = additional_benefits  # Additional benefits included in the
VIP pass

# Class to represent park attractions
class Attraction:
    """Represents an attraction in the theme park."""
    def __init__(self, attraction_id, name, capacity, schedule):
        self.attraction_id = attraction_id  # Unique ID for the attraction
        self.name = name  # Name of the attraction
        self.capacity = capacity  # Maximum capacity of the attraction
        self.schedule = schedule  # Operating schedule of the attraction
        self.guests = []  # Many-to-Many relationship with guests
        self.tickets = []  # Many-to-Many relationship with tickets

    # Add a guest to the attraction
    def add_guest(self, guest):
        self.guests.append(guest)

    # Add a ticket to the attraction
    def add_ticket(self, ticket):
        self.tickets.append(ticket)

# Class to represent purchase orders (Composition with Ticket)
class PurchaseOrder:
    """Represents a purchase order in the ticketing system."""
    def __init__(self, order_id, total_price, order_date):
        self.order_id = order_id  # Unique ID for the purchase order
        self.total_price = total_price  # Total price for all tickets in the order
```

```
        self.order_date = order_date  # Date the order was placed
        self.tickets = []  # List of tickets included in the order (Composition relationship)

    # Method to add a ticket to the purchase order
    def add_ticket(self, ticket):
        self.tickets.append(ticket)

# Class to represent guests (Aggregation with PurchaseOrder)
class Guest:
    """Represents a guest in the theme park system."""
    def __init__(self, name, email, password):
        self.name = name  # Name of the guest
        self.email = email  # Email address of the guest
        self.password = password  # Password for guest's account
        self.purchase_history = []  # List of purchase orders (Aggregation relationship)
        self.attractions = []  # Many-to-Many relationship with attractions

    # Add an attraction to the guest
    def add_attraction(self, attraction):
        self.attractions.append(attraction)

    # Method to add a purchase order to the guest's history
    def add_purchase_order(self, purchase_order):
        self.purchase_history.append(purchase_order)

# Class to represent administrators

class Admin:
    """Represents an admin in the theme park system."""
    def __init__(self, username, password):
        self.username = username  # Username for the admin
        self.password = password  # Password for the admin

    # Method for admin to manage tickets
    def manage_ticket(self, ticket):
        print(f"Admin {self.username} is managing ticket: {ticket.ticket_id}")
```

Test cases

```
if __name__ == "__main__":
```

```
    # Create attractions
    roller_coaster = Attraction("A001", "Roller Coaster", 50, "9 AM - 6 PM")
    ferris_wheel = Attraction("A002", "Ferris Wheel", 30, "10 AM - 8 PM")

    # Create tickets and link to attractions
    vip_ticket = VIPExperiencePass("T006", 550.0, "Valid for one day", 0.0, "Limited
availability", "Reserved seating")
    vip_ticket.add_attraction(roller_coaster)
    vip_ticket.add_attraction(ferris_wheel)

    # Link attractions to the ticket
    roller_coaster.add_ticket(vip_ticket)
    ferris_wheel.add_ticket(vip_ticket)

    # Create a guest and link to attractions
    guest = Guest("Mohammad Abdulrahman", "mohammad.Abdulrahman@zu.com",
"MohdABD@123")
    guest.add_attraction(roller_coaster)
    guest.add_attraction(ferris_wheel)

    # Link guests to attractions
    roller_coaster.add_guest(guest)
    ferris_wheel.add_guest(guest)

    # Print details to verify relationships
    print(f"Guest {guest.name} is visiting the following attractions:")
    for attraction in guest.attractions:
        print(f" - {attraction.name}")

    print(f"The following tickets are valid for {roller_coaster.name}:")
    for ticket in roller_coaster.tickets:
        print(f" - Ticket ID: {ticket.ticket_id}, Type: {ticket.ticket_type}")

    print(f"The following guests visited {ferris_wheel.name}:")
    for guest in ferris_wheel.guests:
        print(f" - Guest Name: {guest.name}")
```

- **The **Graphical User Interface (GUI) **should be intuitive, user-friendly, and visually appealing to enhance the customer experience effectively. Below are key features and functionalities the GUI should incorporate:**

1. ****Account Management: **User Account Creation/Login. Add/Delete/Modify/Display customers' details. View, modify, and delete purchase orders.**

1. Account Management

Features:

- **User Account Creation/Login:**
  - Input fields for user details such as name, email, and password.
  - Buttons for "Login" and "Sign Up."
  - Password recovery options if needed.
- **Add/Delete/Modify/Display Customer Details:**
  - Editable table or list view to display customer details.
  - Buttons for "Add," "Edit," and "Delete" customers.
  - On selecting a customer, an editable form appears to modify their information.
- **View, Modify, and Delete Purchase Orders:**
  - Drop-down or searchable list to view purchase orders associated with a customer.
  - Options to "Modify" or "Delete" orders.
  - Display summary of tickets in each purchase order.

GUI Components:

- **Account Page:**
  - **Fields:** Name, Email, Password, Login/Signup buttons.
  - **Table/List View:** For displaying and managing customers.
  - **Forms:** For adding or editing customer details.
- **Buttons:** Add, Edit, Delete, View Orders.

code:

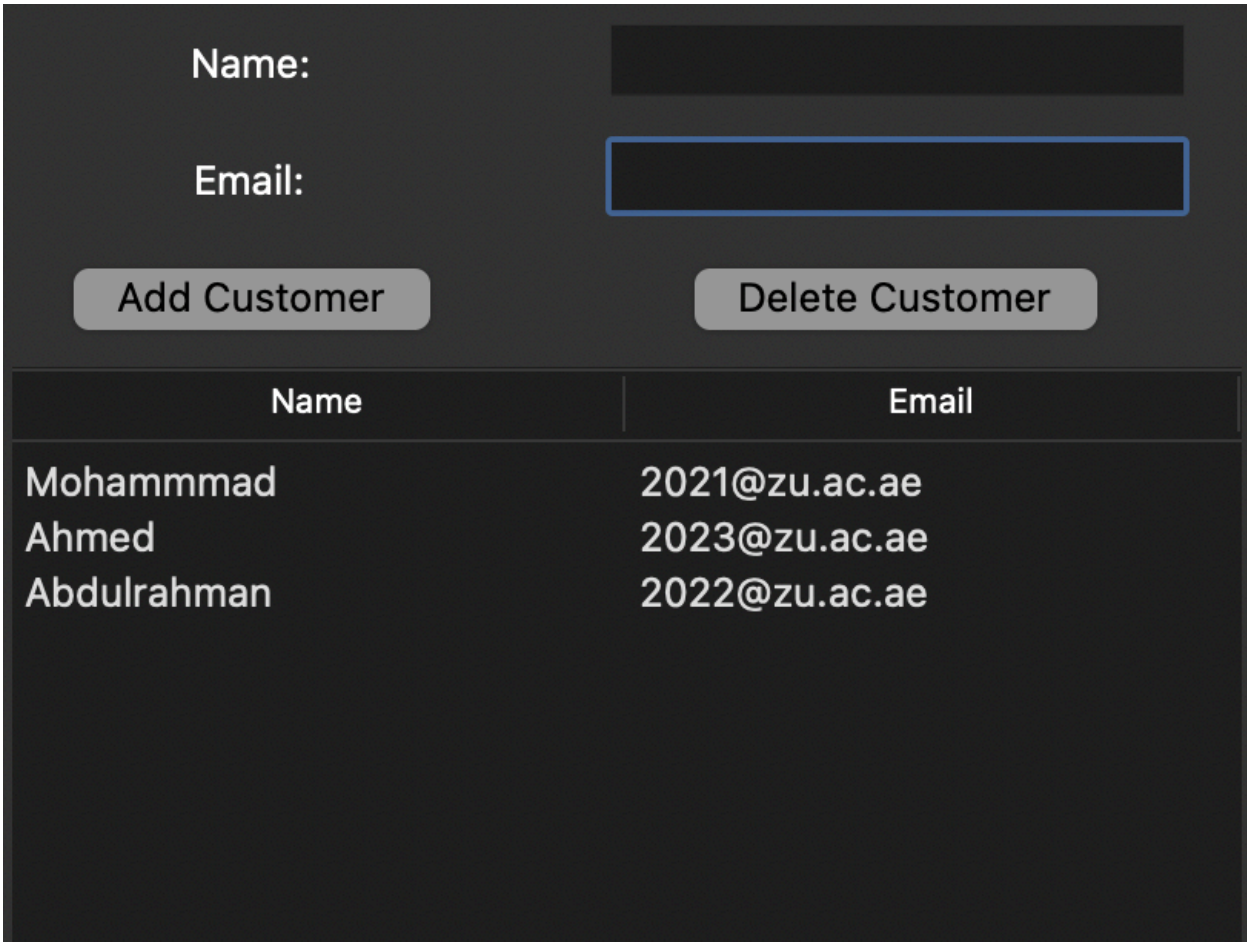
```
import tkinter as tk

from tkinter import messagebox, ttk



# Dummy data storage for customers
```

```
customers = {}



# Functions

def add_customer():

    name = name_entry.get()

    email = email_entry.get()

    if name and email:

        customers[name] = email

        messagebox.showinfo("Success", "Customer added successfully!")

        update_customer_table()

        clear_form()

    else:

        messagebox.showerror("Error", "Name and Email are required!")



def delete_customer():

    selected = customer_table.selection()

    if selected:
```

```
        for item in selected:
            name = customer_table.item(item)['values'][0]
            del customers[name]
        update_customer_table()
        messagebox.showinfo("Success", "Customer deleted successfully!")
    else:
        messagebox.showerror("Error", "Select a customer to delete!")


def update_customer_table():
    customer_table.delete(*customer_table.get_children())
    for name, email in customers.items():
        customer_table.insert("", "end", values=(name, email))


def clear_form():
    name_entry.delete(0, tk.END)
    email_entry.delete(0, tk.END)
```

```
# GUI

root = tk.Tk()

root.title("Account Management")



# Form

tk.Label(root, text="Name:").grid(row=0, column=0, padx=10, pady=5)

name_entry = tk.Entry(root)

name_entry.grid(row=0, column=1, padx=10, pady=5)



tk.Label(root, text="Email:").grid(row=1, column=0, padx=10, pady=5)

email_entry = tk.Entry(root)

email_entry.grid(row=1, column=1, padx=10, pady=5)



add_button = tk.Button(root, text="Add Customer",
command=add_customer)

add_button.grid(row=2, column=0, padx=10, pady=5)



delete_button = tk.Button(root, text="Delete Customer",
command=delete_customer)
```

```
delete_button.grid(row=2, column=1, padx=10, pady=5)



# Customer Table

customer_table = ttk.Treeview(root, columns=("Name", "Email"),
show="headings")

customer_table.heading("Name", text="Name")

customer_table.heading("Email", text="Email")

customer_table.grid(row=3, column=0, columnspan=2, padx=10, pady=5)



root.mainloop()
```



2. **Ticket Purchasing Interface: **Clearly displayed options for various ticket types (e.g., single-day passes, multi-day passes, group discounts). Each ticket type should have essential information: price, validity, and features. The payment interface allows multiple payment methods (credit/debit cards) with clear instructions.

2. Ticket Purchasing Interface

Features:

- Display Ticket Options:
  - Grid or card layout to display all ticket types (e.g., SingleDayTicket, MultiDayTicket).
  - Each card contains:

        - Ticket type (e.g., VIPExperiencePass).
        - Price, validity, and key features.
        - A "Purchase" button.
- Add to Cart:
    - Button on each ticket type to add it to a shopping cart.
    - Cart summary with ticket details, quantity, and total price.
- Payment Interface:
    - Form for credit/debit card details:
        - Card number, expiry date, CVV, and cardholder name.
    - Clear instructions for payment processing.
    - Option to confirm or cancel the purchase.

GUI Components:

2. Ticket Selection Page:
    1. Cards/Grid: Each card displays ticket details.
    2. Buttons: "Purchase" button for each ticket type, and "Add to Cart."
    3. Cart Summary: Sidebar or pop-up displaying added tickets and total price.
3. Payment Page:
    1. Fields: Credit/debit card input fields.
    2. Buttons: Confirm Payment, Cancel.

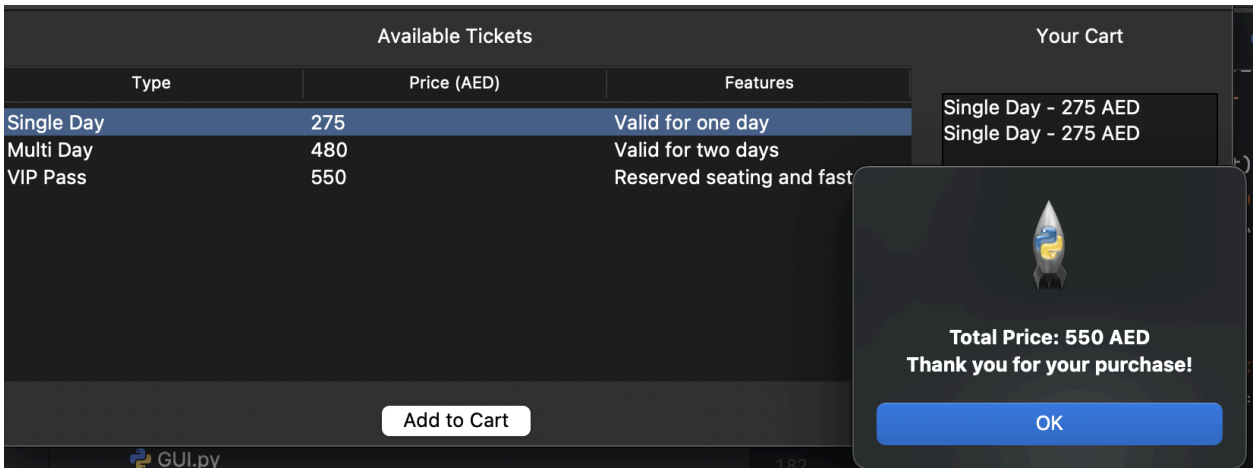
Code:

```
# Ticket Purchasing GUI


import tkinter as tk

from tkinter import ttk, messagebox


# Dummy data for tickets

tickets = [

    {"type": "Single Day", "price": 275, "features": "Valid for one day"},

    {"type": "Multi Day", "price": 480, "features": "Valid for two days"},
```

```
    {"type": "VIP Pass", "price": 550, "features": "Reserved seating and fast access"}
]

# Cart
cart = []

# Functions
def add_to_cart(ticket):
    cart.append(ticket)
    update_cart()

def update_cart():
    cart_list.delete(0, tk.END)
    for ticket in cart:
        cart_list.insert(tk.END, f"{ticket['type']} - {ticket['price']} AED")

def checkout():
    if not cart:
        messagebox.showerror("Error", "Your cart is empty!")
    else:
        total = sum(ticket['price'] for ticket in cart)
        messagebox.showinfo("Checkout", f"Total Price: {total} AED\nThank you for your purchase!")
        cart.clear()
        update_cart()
```

```
# GUI

root = tk.Tk()

root.title("Ticket Purchasing Interface")


# Ticket Options

tk.Label(root, text="Available Tickets").grid(row=0, column=0, padx=10, pady=5)

ticket_table = ttk.Treeview(root, columns=("Type", "Price", "Features"),
show="headings")

ticket_table.heading("Type", text="Type")

ticket_table.heading("Price", text="Price (AED)")

ticket_table.heading("Features", text="Features")

ticket_table.grid(row=1, column=0, padx=10, pady=5)


# Populate ticket table

for ticket in tickets:

    ticket_table.insert("", "end", values=(ticket["type"], ticket["price"],
ticket["features"]))


# Add to Cart Button

add_button = tk.Button(root, text="Add to Cart", command=lambda:
add_to_cart(tickets[0]))

add_button.grid(row=2, column=0, padx=10, pady=5)


# Cart

tk.Label(root, text="Your Cart").grid(row=0, column=1, padx=10, pady=5)
```

```
cart_list = tk.Listbox(root)
cart_list.grid(row=1, column=1, padx=10, pady=5)

# Checkout Button
checkout_button = tk.Button(root, text="Checkout", command=checkout)
checkout_button.grid(row=2, column=1, padx=10, pady=5)

root.mainloop()
```



3. ****Admin Dashboard: **This dashboard displays ticket sales (the number of tickets sold per day) and provides access to modify discount availability.**

3. Admin Dashboard

Features:

- Display Ticket Sales:
  - Table or chart showing the number of tickets sold daily, categorized by ticket type.

    - Options to filter data by date or ticket type.
- Modify Discount Availability:
    - List of all available discounts with checkboxes or toggles to enable/disable them.
    - Editable fields to change discount percentages or conditions.

GUI Components:

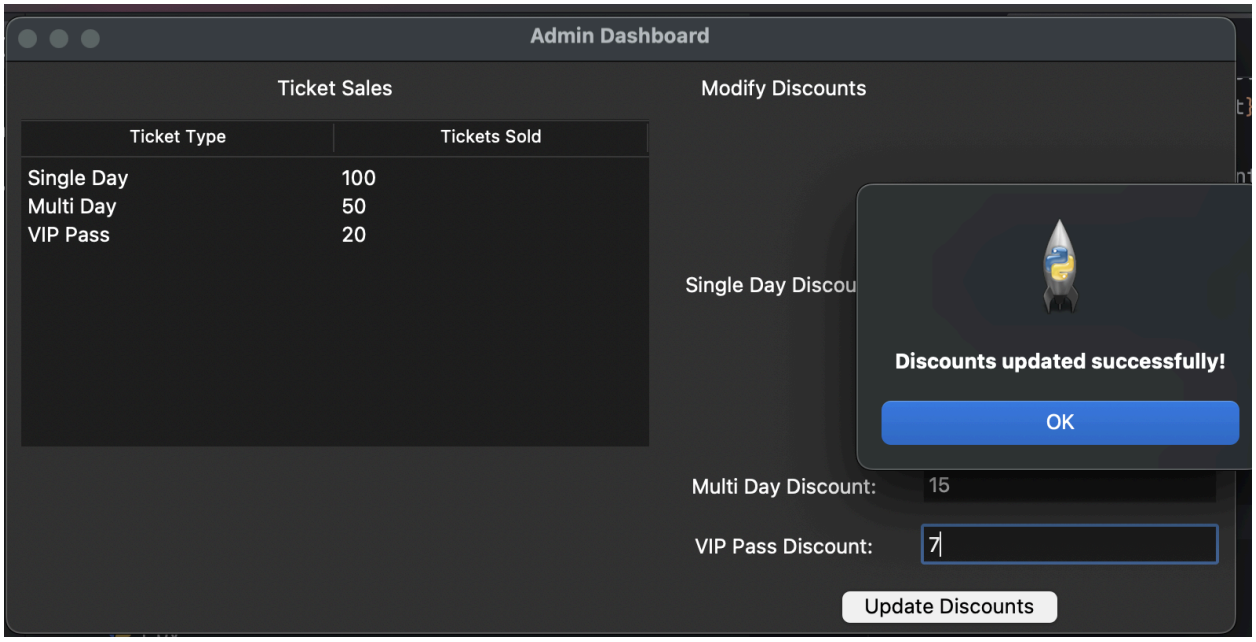
- Dashboard Page:
    - Ticket Sales Section: Charts (e.g., bar or line chart) and tables for ticket sales data.
    - Discount Management Section: List view with toggles for discount activation and editable fields for discount details.
- Buttons: Save Changes, Reset.

Code:

```
# Admin Dashboard


import tkinter as tk

from tkinter import ttk, messagebox


# Dummy data

sales_data = [

    {"type": "Single Day", "sales": 100},

    {"type": "Multi Day", "sales": 50},

    {"type": "VIP Pass", "sales": 20}

]

discounts = {"Single Day": 0, "Multi Day": 10, "VIP Pass": 5}


# Functions

def update_discounts():
```

```
    for ticket in discounts:
        discounts[ticket] = int(discount_entries[ticket].get())
    messagebox.showinfo("Success", "Discounts updated successfully!")

# GUI
root = tk.Tk()
root.title("Admin Dashboard")

# Ticket Sales
tk.Label(root, text="Ticket Sales").grid(row=0, column=0, padx=10, pady=5)
sales_table = ttk.Treeview(root, columns=("Type", "Sales"), show="headings")
sales_table.heading("Type", text="Ticket Type")
sales_table.heading("Sales", text="Tickets Sold")
sales_table.grid(row=1, column=0, padx=10, pady=5)

# Populate sales table
for data in sales_data:
    sales_table.insert("", "end", values=(data["type"], data["sales"]))

# Discounts
tk.Label(root, text="Modify Discounts").grid(row=0, column=1, padx=10, pady=5)
discount_entries = {}
row = 1
for ticket, discount in discounts.items():
    tk.Label(root, text=f"{ticket} Discount:").grid(row=row, column=1, padx=10, pady=5)
```

```
    entry = tk.Entry(root)
    entry.insert(0, discount)
    entry.grid(row=row, column=2, padx=10, pady=5)
    discount_entries[ticket] = entry
    row += 1

# Update Discounts Button
update_button = tk.Button(root, text="Update Discounts", command=update_discounts)
update_button.grid(row=row, column=1, columnspan=2, padx=10, pady=5)

root.mainloop()
```



Flow of the GUI

1. Login/Signup Screen:
    - Users can log in or create an account.
2. Home Screen:
    - Displays available tickets with a shopping cart summary.
    - Navigation options to "Account Management" or "Purchase Tickets."
3. Admin Dashboard:
    - Admins log in and access analytics, sales data, and discount settings.

- **All details must be stored in **binary files using the Pickle library **in Python. Multiple binary files may be used to store related information.**
    1. **Account Management GUI**:
    - **Binary File:** `customers.pkl`
    - Stores customer details in a binary file and retrieves them on startup.

```
import tkinter as tk

from tkinter import messagebox, ttk

import pickle



# Binary file for storing customer data

CUSTOMER_FILE = "customers.pkl"



# Load customers from binary file
```

```
def load_customers():
    try:
        with open(CUSTOMER_FILE, "rb") as file:
            return pickle.load(file)
    except (FileNotFoundError, EOFError):
        return {}

# Save customers to binary file
def save_customers():
    with open(CUSTOMER_FILE, "wb") as file:
        pickle.dump(customers, file)

# Functions
def add_customer():
    name = name_entry.get()
    email = email_entry.get()
```

```
    if name and email:
        customers[name] = email
        save_customers()
        messagebox.showinfo("Success", "Customer added successfully!")
        update_customer_table()
        clear_form()
    else:
        messagebox.showerror("Error", "Name and Email are required!")


def delete_customer():
    selected = customer_table.selection()
    if selected:
        for item in selected:
            name = customer_table.item(item)['values'][0]
            del customers[name]
        save_customers()
```

```
        update_customer_table()
        messagebox.showinfo("Success", "Customer deleted successfully!")
    else:
        messagebox.showerror("Error", "Select a customer to delete!")

def update_customer_table():
    customer_table.delete(*customer_table.get_children())
    for name, email in customers.items():
        customer_table.insert("", "end", values=(name, email))

def clear_form():
    name_entry.delete(0, tk.END)
    email_entry.delete(0, tk.END)

# GUI
root = tk.Tk()
```

```
root.title("Account Management")



# Load customers

customers = load_customers()



# Form

tk.Label(root, text="Name:").grid(row=0, column=0, padx=10, pady=5)

name_entry = tk.Entry(root)

name_entry.grid(row=0, column=1, padx=10, pady=5)



tk.Label(root, text="Email:").grid(row=1, column=0, padx=10, pady=5)

email_entry = tk.Entry(root)

email_entry.grid(row=1, column=1, padx=10, pady=5)



add_button = tk.Button(root, text="Add Customer", command=add_customer)

add_button.grid(row=2, column=0, padx=10, pady=5)
```

```
delete_button = tk.Button(root, text="Delete Customer", command=delete_customer)

delete_button.grid(row=2, column=1, padx=10, pady=5)



# Customer Table

customer_table = ttk.Treeview(root, columns=("Name", "Email"), show="headings")

customer_table.heading("Name", text="Name")

customer_table.heading("Email", text="Email")

customer_table.grid(row=3, column=0, columnspan=2, padx=10, pady=5)



update_customer_table()



root.mainloop()
```

2.Ticket Purchasing Interface GUI:

- **Binary File:** `cart.pkl`
- Stores cart details in a binary file for persistence.

```
# Ticket Purchasing GUI with Binary Storage

import tkinter as tk

from tkinter import ttk, messagebox

import pickle



# Binary file for storing cart data

CART_FILE = "cart.pkl"



# Dummy data for tickets

tickets = [

    {"type": "Single Day", "price": 275, "features": "Valid for one day"},

    {"type": "Multi Day", "price": 480, "features": "Valid for two days"},

    {"type": "VIP Pass", "price": 550, "features": "Reserved seating and fast access"}

]



# Load cart from binary file
```

```
def load_cart():
    try:
        with open(CART_FILE, "rb") as file:
            return pickle.load(file)
    except (FileNotFoundError, EOFError):
        return []

# Save cart to binary file
def save_cart():
    with open(CART_FILE, "wb") as file:
        pickle.dump(cart, file)

# Cart
cart = load_cart()

# Functions
```

```
def add_to_cart(ticket):
    cart.append(ticket)
    save_cart()
    update_cart()


def update_cart():
    cart_list.delete(0, tk.END)
    for ticket in cart:
        cart_list.insert(tk.END, f"{ticket['type']} - {ticket['price']} AED")


def checkout():
    if not cart:
        messagebox.showerror("Error", "Your cart is empty!")
    else:
        total = sum(ticket['price'] for ticket in cart)
        messagebox.showinfo("Checkout", f"Total Price: {total} AED\nThank you for your purchase!")
```

```
        cart.clear()

        save_cart()

        update_cart()



# GUI

root = tk.Tk()

root.title("Ticket Purchasing Interface")



# Ticket Options

tk.Label(root, text="Available Tickets").grid(row=0, column=0, padx=10, pady=5)

ticket_table = ttk.Treeview(root, columns=("Type", "Price", "Features"),
show="headings")

ticket_table.heading("Type", text="Type")

ticket_table.heading("Price", text="Price (AED)")

ticket_table.heading("Features", text="Features")

ticket_table.grid(row=1, column=0, padx=10, pady=5)
```

```
# Populate ticket table

for ticket in tickets:

    ticket_table.insert("", "end", values=(ticket["type"], ticket["price"], ticket["features"]))



# Add to Cart Button

add_button = tk.Button(root, text="Add to Cart", command=lambda:
add_to_cart(tickets[0]))

add_button.grid(row=2, column=0, padx=10, pady=5)



# Cart

tk.Label(root, text="Your Cart").grid(row=0, column=1, padx=10, pady=5)

cart_list = tk.Listbox(root)

cart_list.grid(row=1, column=1, padx=10, pady=5)



# Checkout Button

checkout_button = tk.Button(root, text="Checkout", command=checkout)

checkout_button.grid(row=2, column=1, padx=10, pady=5)
```

update_cart()

root.mainloop()

3. Admin Dashboard GUI:

- **Binary File:** `sales.pkl`
- Stores ticket sales and discount details in binary files.

```
# Admin Dashboard with Binary Storage

import tkinter as tk

from tkinter import ttk, messagebox

import pickle


# Binary files for storing sales and discounts

SALES_FILE = "sales.pkl"

DISCOUNTS_FILE = "discounts.pkl"


# Dummy data

default_sales = [

    {"type": "Single Day", "sales": 100},

    {"type": "Multi Day", "sales": 50},
```

```
    {"type": "VIP Pass", "sales": 20}
]
default_discounts = {"Single Day": 0, "Multi Day": 10, "VIP Pass": 5}

# Load data from binary files
def load_data(file, default):
    try:
        with open(file, "rb") as f:
            return pickle.load(f)
    except (FileNotFoundError, EOFError):
        return default

def save_data(file, data):
    with open(file, "wb") as f:
        pickle.dump(data, f)

# Load sales and discounts
sales_data = load_data(SALES_FILE, default_sales)
discounts = load_data(DISCOUNTS_FILE, default_discounts)

# Functions
def update_discounts():
    for ticket in discounts:
```

```
        discounts[ticket] = int(discount_entries[ticket].get())
    save_data(DISCOUNTS_FILE, discounts)
    messagebox.showinfo("Success", "Discounts updated successfully!")

# GUI
root = tk.Tk()
root.title("Admin Dashboard")

# Ticket Sales
tk.Label(root, text="Ticket Sales").grid(row=0, column=0, padx=10, pady=5)
sales_table = ttk.Treeview(root, columns=("Type", "Sales"), show="headings")
sales_table.heading("Type", text="Ticket Type")
sales_table.heading("Sales", text="Tickets Sold")
sales_table.grid(row=1, column=0, padx=10, pady=5)

# Populate sales table
for data in sales_data:
    sales_table.insert("", "end", values=(data["type"], data["sales"]))

# Discounts
tk.Label(root, text="Modify Discounts").grid(row=0, column=1, padx=10, pady=5)
discount_entries = {}
row = 1
```

```
for ticket, discount in discounts.items():
    tk.Label(root, text=f"{ticket} Discount:").grid(row=row, column=1, padx=10, pady=5)
    entry = tk.Entry(root)
    entry.insert(0, discount)
    entry.grid(row=row, column=2, padx=10, pady=5)
    discount_entries[ticket] = entry
    row += 1

# Update Discounts Button
update_button = tk.Button(root, text="Update Discounts", command=update_discounts)
update_button.grid(row=row, column=1, columnspan=2, padx=10, pady=5)

root.mainloop()
```

- **All **scenarios must be tested **to ensure all requirements are met. Explain the different types of testing that was done to ensure that all requirements were met.**

Testing Type	Purpose	Example
Unit Testing	Test individual components in isolation.	Add a customer and check if they appear in the table.
Integration Testing	Test parts working together.	Link tickets to a purchase order and

		check if it shows in the guest's history.
System Testing	Test the full system end-to-end.	A guest creates an account, buys tickets, and checks out successfully.
Usability Testing	Check if the GUI is easy to use.	Test the admin dashboard for clear sales and discount options.
Functional Testing	Check if features work as required.	Add tickets to the cart, apply discounts, and confirm total price is correct.
Performance Testing	Test system with large data.	Add 1,000 customers and check if the system stays fast.
Regression Testing	Ensure old features still work.	Apply new discounts and confirm old customer data is not lost.
Security Testing	Ensure data is safe.	Try editing binary files and check if the system handles errors correctly.

Summary of learnings:

Abdulrahman:

I learned how to combine features like account management, ticket purchasing, and admin tasks into a single program. I practiced writing modular code by breaking the system into smaller parts, making it easier to manage and reuse. I also gained skills in using Pickle to save data in binary files, ensuring security and reliability. By testing the program for various scenarios, such as adding customers, checking out tickets, and changing discounts, I understood the importance of thorough testing in real-world applications.

Mohammad:

I learned how to create and use UML diagrams to represent relationships such as inheritance, aggregation, and composition. I also understood how to translate these diagrams into functional Python code that interacts with data effectively. By practicing with the Pickle library, I gained insights into saving and loading data from binary files, emphasizing the importance of data persistence. Additionally, I designed a user-friendly GUI using Tkinter to manage accounts, buy tickets, and navigate an admin dashboard. Finally, I developed skills in testing and debugging to ensure smooth and reliable functionality in my applications.

GitHuB Link

https://github.com/MohammadAbdullahHashim/220ASS3.git