**Final Assignment Programming Fundamentals**

Maitha Abdulla

Shatha Khaled

Ahmed Waleed

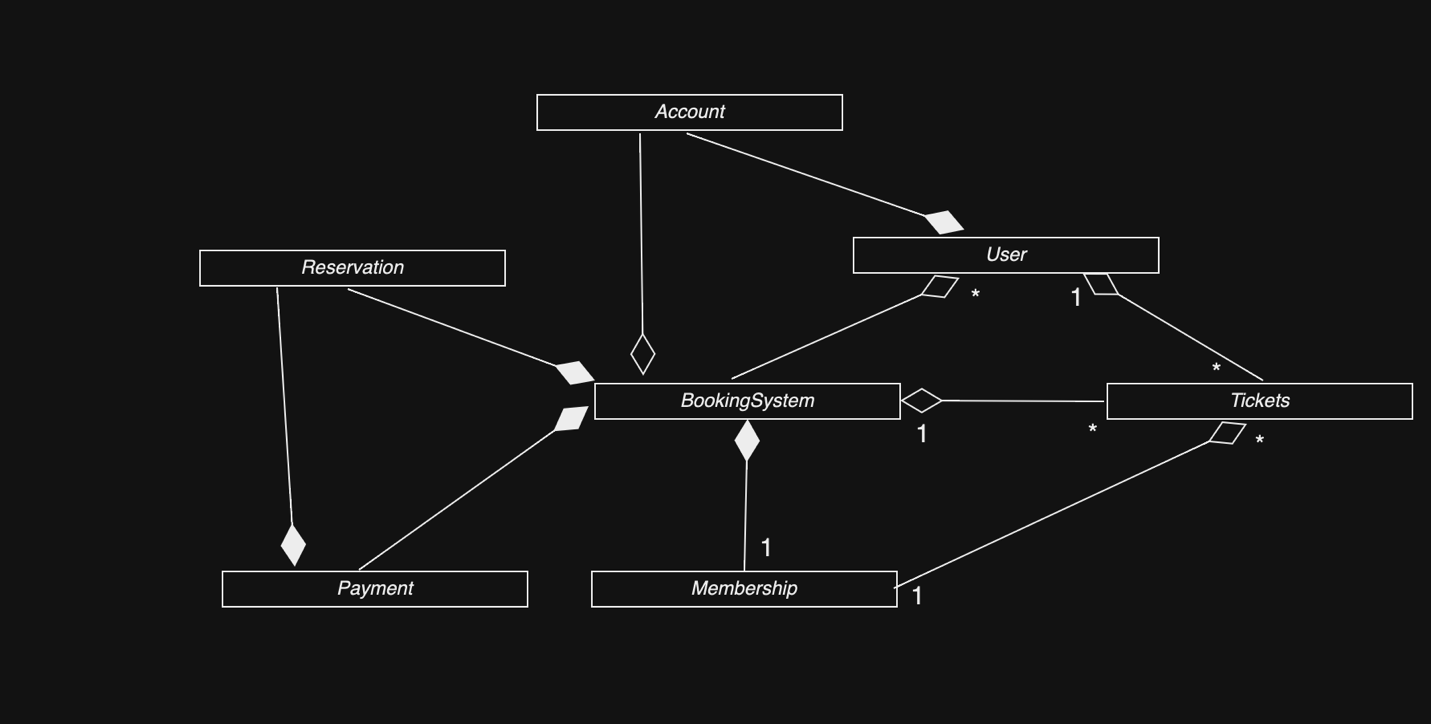
College of Interdisciplinary Studies, Zayed University

ICS220-22527: Program. Fund.

Professor: Sujith Mathew

December 5, 2024

UML Class Diagram:



A screenshot of a computer screen

Description automatically generated

**Description:**

**Account and User**

* **relationship**: it is a composition because an account is a part of a user. If the user was not there, there wont be an account. If the account was deleted, the user will not appear in the system. Every user has one account
* **Assumption**: Users can log in or manage their details through an account

**Account and BookingSystem**

* **relationship**: it is an aggregation because a booking system can still exist if the account was deleted and every booking system can have multiple accounts in it
* **Assumption**: Accounts are created and managed exclusively by the booking system

**BookingSystem and User**

* **relationship**: it is an aggregation because a user can exist independently if the booking system was deleted and every booking system can have multiple users
* **Assumption**: The booking system does not own the users but keeps track of their information

**BookingSystem and Tickets**

* **relationship**: it is an aggregation because a ticket can exist independently if the booking system was deleted because the user can purchase a ticket physically and every booking system can have multiple tickets
* **Assumption**: All tickets are issued and maintained by the booking system

**User and Tickets**

* **relationship**: it is an aggregation because a ticket can exist independently in the booking system even if there were no users and every user can purchase multiple tickets
* **Assumption**: Users need tickets to access park services, and tickets are stored under their accounts

**Reservation and BookingSystem**

* **relationship**: it is a composition because a reservation cant exist independently if there wasn’t any booking system and a booking system can have multiple reservations from different users
* **Assumption**: Reservations are created, tracked, and stored entirely by the system

**Reservation and Payment**

* **relationship**: it is a composition because a payment cant be done if there wasn’t a reservation so payments cannot exist independently and every reservation has one payment process
* **Assumption**: Payments are processed for reservations but are stored separately in the system

**Membership and BookingSystem**

* **relationship**: it is a composition because memberships are part of the booking system and cannot exist without it and the booking system can assign one membership to each user if they wanted
* **Assumption**: The system fully manages memberships, including creating, renewing, and tracking them

**User and BookingSystem**

* **relationship**: it is a composition because memberships are part of the booking system and cannot exist without it and the booking system can assign one membership to each user if they wanted
* **Assumption**: The system fully manages memberships, including creating, renewing, and tracking them

**Membership and Ticket**

* **relationship**: it is an aggregation because a ticket can exist independently without a membership and each ticket can have one membership discount applied to it
* **Assumption**: Memberships may allow members to receive specific types of tickets such as free or discounted

**Payment and BookingSystem**

* **relationship**: it is a composition because a payment cannot be done if there was no booking system to manage it and the system generates one payment for each reservation
* **Assumption**: The BookingSystem processes payments for tickets, memberships, or reservations

**Defining the UML Class Diagram with codes:**

class User:

"""Class representing a system User"""

def \_\_init\_\_(self, name, email, phone\_number, membership\_status, booking\_history):

self.\_uName = name

self.\_uEmail = email

self.\_uPhoneNumber = phone\_number

self.\_uMembershipStatus = membership\_status

self.\_uBookingHistory = booking\_history

def set\_name(self, name):

self.\_uName = name

def get\_name(self):

return self.\_uName

def set\_email(self, email):

self.\_uEmail = email

def get\_email(self):

return self.\_uEmail

def set\_phone\_number(self, phone\_number):

self.\_uPhoneNumber = phone\_number

def get\_phone\_number(self):

return self.\_uPhoneNumber

def set\_membership\_status(self, status):

self.\_uMembershipStatus = status

def get\_membership\_status(self):

return self.\_uMembershipStatus

def set\_booking\_history(self, history):

self.\_uBookingHistory = history

def get\_booking\_history(self):

return self.\_uBookingHistory

def get\_details(self):

return f"User: {self.\_uName}, Email: {self.\_uEmail}, Membership: {self.\_uMembershipStatus}"

class Account:

"""Class representing a User Account"""

def \_\_init\_\_(self, account\_id, username, password, balance, created\_at):

self.\_aAccountId = account\_id

self.\_aUsername = username

self.\_aPassword = password

self.\_aBalance = balance

self.\_aCreatedAt = created\_at

def set\_account\_id(self, account\_id):

self.\_aAccountId = account\_id

def get\_account\_id(self):

return self.\_aAccountId

def set\_username(self, username):

self.\_aUsername = username

def get\_username(self):

return self.\_aUsername

def set\_password(self, password):

self.\_aPassword = password

def get\_password(self):

return self.\_aPassword

def set\_balance(self, balance):

self.\_aBalance = balance

def get\_balance(self):

return self.\_aBalance

def set\_created\_at(self, created\_at):

self.\_aCreatedAt = created\_at

def get\_created\_at(self):

return self.\_aCreatedAt

def get\_details(self):

return f"Account {self.\_aAccountId}, Username: {self.\_aUsername}, Balance: {self.\_aBalance}"

class BookingSystem:

"""Class representing a Booking System"""

def \_\_init\_\_(self, system\_id, available\_tickets, reservations, capacity, status):

self.\_bSystemId = system\_id

self.\_bAvailableTickets = available\_tickets

self.\_bReservations = reservations

self.\_bCapacity = capacity

self.\_bSystemStatus = status

def set\_system\_id(self, system\_id):

self.\_bSystemId = system\_id

def get\_system\_id(self):

return self.\_bSystemId

def set\_available\_tickets(self, tickets):

self.\_bAvailableTickets = tickets

def get\_available\_tickets(self):

return self.\_bAvailableTickets

def set\_reservations(self, reservations):

self.\_bReservations = reservations

def get\_reservations(self):

return self.\_bReservations

def set\_capacity(self, capacity):

self.\_bCapacity = capacity

def get\_capacity(self):

return self.\_bCapacity

def set\_system\_status(self, status):

self.\_bSystemStatus = status

def get\_system\_status(self):

return self.\_bSystemStatus

def get\_details(self):

return f"System ID: {self.\_bSystemId}, Status: {self.\_bSystemStatus}, Capacity: {self.\_bCapacity}"

class Tickets:

"""Class representing Tickets"""

def \_\_init\_\_(self, ticket\_id, ticket\_type, price, validity\_date, event\_name):

self.\_tId = ticket\_id

self.\_tType = ticket\_type

self.\_tPrice = price

self.\_tValidityDate = validity\_date

self.\_tEventName = event\_name

def set\_ticket\_id(self, ticket\_id):

self.\_tId = ticket\_id

def get\_ticket\_id(self):

return self.\_tId

def set\_type(self, ticket\_type):

self.\_tType = ticket\_type

def get\_type(self):

return self.\_tType

def set\_price(self, price):

self.\_tPrice = price

def get\_price(self):

return self.\_tPrice

def set\_validity\_date(self, validity\_date):

self.\_tValidityDate = validity\_date

def get\_validity\_date(self):

return self.\_tValidityDate

def set\_event\_name(self, event\_name):

self.\_tEventName = event\_name

def get\_event\_name(self):

return self.\_tEventName

def get\_details(self):

return f"Ticket ID: {self.\_tId}, Event: {self.\_tEventName}, Price: {self.\_tPrice}"

class Reservation:

"""Class representing a Reservation"""

def \_\_init\_\_(self, reservation\_id, user\_id, ticket\_id, date, status):

self.\_rId = reservation\_id

self.\_rUserId = user\_id

self.\_rTicketId = ticket\_id

self.\_rDate = date

self.\_rStatus = status

def set\_reservation\_id(self, reservation\_id):

self.\_rId = reservation\_id

def get\_reservation\_id(self):

return self.\_rId

def set\_user\_id(self, user\_id):

self.\_rUserId = user\_id

def get\_user\_id(self):

return self.\_rUserId

def set\_ticket\_id(self, ticket\_id):

self.\_rTicketId = ticket\_id

def get\_ticket\_id(self):

return self.\_rTicketId

def set\_date(self, date):

self.\_rDate = date

def get\_date(self):

return self.\_rDate

def set\_status(self, status):

self.\_rStatus = status

def get\_status(self):

return self.\_rStatus

def get\_details(self):

return f"Reservation {self.\_rId}, Status: {self.\_rStatus}, Date: {self.\_rDate}"

class Membership:

"""Class representing a Membership"""

def \_\_init\_\_(self, membership\_id, membership\_type, discount\_rate, validity\_period, user\_id):

self.\_mId = membership\_id

self.\_mType = membership\_type

self.\_mDiscountRate = discount\_rate

self.\_mValidityPeriod = validity\_period

self.\_mUserId = user\_id

def set\_membership\_id(self, membership\_id):

self.\_mId = membership\_id

def get\_membership\_id(self):

return self.\_mId

def set\_membership\_type(self, membership\_type):

self.\_mType = membership\_type

def get\_membership\_type(self):

return self.\_mType

def set\_discount\_rate(self, discount\_rate):

self.\_mDiscountRate = discount\_rate

def get\_discount\_rate(self):

return self.\_mDiscountRate

def set\_validity\_period(self, validity\_period):

self.\_mValidityPeriod = validity\_period

def get\_validity\_period(self):

return self.\_mValidityPeriod

def set\_user\_id(self, user\_id):

self.\_mUserId = user\_id

def get\_user\_id(self):

return self.\_mUserId

def get\_details(self):

return f"Membership {self.\_mId}, Type: {self.\_mType}, Discount: {self.\_mDiscountRate}%"

class Payment:

"""Class representing a Payment"""

def \_\_init\_\_(self, payment\_id, amount, method, date, status):

self.\_pId = payment\_id

self.\_pAmount = amount

self.\_pMethod = method

self.\_pDate = date

self.\_pStatus = status

def set\_payment\_id(self, payment\_id):

self.\_pId = payment\_id

def get\_payment\_id(self):

return self.\_pId

def set\_amount(self, amount):

self.\_pAmount = amount

def get\_amount(self):

return self.\_pAmount

def set\_method(self, method):

self.\_pMethod = method

def get\_method(self):

return self.\_pMethod

def set\_date(self, date):

self.\_pDate = date

def get\_date(self):

return self.\_pDate

def set\_status(self, status):

self.\_pStatus = status

def get\_status(self):

return self.\_pStatus

def get\_details(self):

return f"Payment {self.\_pId}, Amount: {self.\_pAmount}, Status: {self.\_pStatus}"

**Python code for pickle and multiple files:**

import pickle

from datetime import datetime

# Helper function to save data using Pickle

def saveFile(data, filename):

with open(filename, 'wb') as file:

pickle.dump(data, file)

# Helper function to load data using Pickle

def loadFile(filename):

try:

with open(filename, 'rb') as file:

return pickle.load(file)

except FileNotFoundError:

return []

class User:

def \_\_init\_\_(self, uName, uEmail, uPhoneNumber):

self.uName = uName

self.uEmail = uEmail

self.uPhoneNumber = uPhoneNumber

self.uMembershipStatus = False

self.uBookingHistory = []

def register(self):

print(f"{self.uName} has been registered successfully.")

return self

def login(self, email):

if email == self.uEmail:

print(f"{self.uName} has logged in.")

return True

raise ValueError("Invalid email address provided!")

class Account:

def \_\_init\_\_(self, aAccountId, aUsername, aPassword):

self.aAccountId = aAccountId

self.aUsername = aUsername

self.aPassword = aPassword

self.aBalance = 0.0

self.aCreatedAt = datetime.now().strftime('%Y-%m-%d %H:%M:%S')

def createAccount(self):

print(f"Account for {self.aUsername} has been created.")

def login(self, username, password):

if username == self.aUsername and password == self.aPassword:

print(f"{self.aUsername} has logged in.")

return True

raise ValueError("Invalid username or password provided!")

def checkBalance(self):

return self.aBalance

class BookingSystem:

def \_\_init\_\_(self, bSystemId, bCapacity):

self.bSystemId = bSystemId

self.bAvailableTickets = []

self.bReservations = []

self.bCapacity = bCapacity

self.bSystemStatus = "No more tickets available!"

def addTicket(self, ticket):

self.bAvailableTickets.append(ticket)

saveFile(self.bAvailableTickets, 'tickets.pkl')

print(f"Ticket #{ticket.tId} booked.")

def makeReservation(self, user, ticket\_id):

if len(self.bReservations) >= self.bCapacity:

return self.bSystemStatus

ticket = None

# Keep iterating through all available tickets until a match is found

for t in self.bAvailableTickets:

if t.tId == ticket\_id:

ticket = t

break

if not ticket:

raise Exception("Ticket not found!")

reservation = {"user": user.uName, "ticket\_id": ticket\_id, "date": str(datetime.now())}

self.bReservations.append(reservation)

saveFile(self.bReservations, 'reservations.pkl')

return f"Reservation made for {user.uName}."

def checkAvailability(self):

if len(self.bReservations) < self.bCapacity:

self.bSystemStatus = "Available"

return True

return False

class Tickets:

def \_\_init\_\_(self, tId, tType, tPrice, tValidityDate, tEventName):

self.tId = tId

self.tType = tType

self.tPrice = tPrice

self.tValidityDate = tValidityDate

self.tEventName = tEventName

def updatePrice(self, new\_price):

self.tPrice = new\_price

print(f"Ticket #{self.tId} price is now {self.tPrice:.2f}.")

def isValid(self):

return datetime.now() <= self.tValidityDate

def applyDiscount(self, discount\_rate):

new\_price = self.tPrice - self.tPrice \* discount\_rate

print(f"Discount applied! New price: {new\_price:.2f}")

return new\_price

class Reservation:

def \_\_init\_\_(self, rId, rUserId, rTicketId, rDate):

self.rId = rId

self.rUserId = rUserId

self.rTicketId = rTicketId

self.rDate = rDate

self.rStatus = "Pending"

def confirmReservation(self):

self.rStatus = "Confirmed"

print(f"Reservation #{self.rId} has been confirmed.")

def cancelReservation(self):

self.rStatus = "Cancelled"

print(f"Reservation #{self.rId} has been cancelled.")

return True

def updateReservation(self, new\_date):

self.rDate = new\_date

print(f"Reservation #{self.rId} date is now {self.rDate}.")

def isActive(self):

return self.rStatus == "Confirmed"

class Membership:

def \_\_init\_\_(self, mId, mType, mDiscountRate, mValidityPeriod, mUserId):

self.mId = mId

self.mType = mType

self.mDiscountRate = mDiscountRate

self.mValidityPeriod = mValidityPeriod

self.mUserId = mUserId

self.mStatus = "Pending"

def activateMembership(self):

self.mStatus = "Active"

print(f"Membership #{self.mId} has been activated.")

def renewMembership(self):

print(f"Membership #{self.mId} has been renewed for {self.mValidityPeriod} days.")

def applyDiscount(self, ticket):

ticket.applyDiscount(self.mDiscountRate)

def cancelMembership(self):

self.mStatus = "Cancelled"

print(f"Membership #{self.mId} has been cancelled.")

class Payment:

def \_\_init\_\_(self, pId, pAmount, pMethod):

self.pId = pId

self.pAmount = pAmount

self.pMethod = pMethod

self.pDate = datetime.now().strftime('%Y-%m-%d %H:%M:%S')

self.pStatus = "Pending"

def processPayment(self):

self.pStatus = "Completed"

print(f"Payment #{self.pId} of amount {self.pAmount:.2f} has been processed.")

return True

def refundPayment(self):

self.pStatus = "Refunded"

print(f"Payment #{self.pId} of amount {self.pAmount:.2f} has been refunded.")

return True

def getPaymentDetails(self):

return {

"Payment ID": self.pId,

"Amount": self.pAmount,

"Method": self.pMethod,

"Date": self.pDate,

"Status": self.pStatus

}

class AdminDashboard:

def \_\_init\_\_(self, booking\_system):

self.booking\_system = booking\_system

def displaySummary(self):

print("Admin Dashboard Summary:")

print(f"Total Reservations: {len(self.booking\_system.bReservations)}")

print(f"Total Tickets Available: {len(self.booking\_system.bAvailableTickets)}")

print(f"System Capacity: {self.booking\_system.bCapacity}")

def testCases():

user = User("John Doe", "john.doe@example.com", "+1 437 669-0171")

user.register()

user.login("john.doe@example.com")

account = Account("A1", "johndoe", "password123")

account.createAccount()

account.login(account.aUsername, account.aPassword)

print("Account balance:", account.checkBalance())

system = BookingSystem("B1", 10)

ticket = Tickets("T1", "Standard", 50.0, datetime(2024, 12, 31), "Concert")

print("Is ticket valid?", ticket.isValid())

ticket.applyDiscount(0.2)

system.addTicket(ticket)

system.makeReservation(user, ticket.tId)

admin = AdminDashboard(system)

admin.displaySummary()

reservation = Reservation("R1", user.uName, ticket.tId, datetime.now())

reservation.confirmReservation()

membership = Membership("M1", "Gold", 0.1, 365, user.uName)

membership.activateMembership()

membership.applyDiscount(ticket)

payment = Payment("P1", 45.0, "Credit Card")

payment.processPayment()

print("Payment details:", payment.getPaymentDetails())

if \_\_name\_\_ == "\_\_main\_\_":

testCases()

**Explanation for the codes:**

The test case above is checking each of the parts, like accounts, tickets, and payments, to make sure that they work on their own and it also tests how they work together. Where it also ensures that the features like reservations, discounts and admin summaries do what they are supposed to do.

**GUI Python codes:**

import tkinter as tk

from tkinter import messagebox, ttk

class ThemeParkGUI:

def \_init\_(self, root):

self.root = root

self.root.title("Adventure Land Theme Park")

self.root.geometry("800x600")

# Mock database for accounts and purchase orders

self.accounts = {}

self.purchase\_orders = {}

# Initialize frames

self.init\_gui()

def init\_gui(self):

# Tabs for different sections

tab\_control = ttk.Notebook(self.root)

# Account Management Tab

self.account\_frame = ttk.Frame(tab\_control)

tab\_control.add(self.account\_frame, text="Account Management")

self.setup\_account\_management()

# Ticket Purchasing Tab

self.ticket\_frame = ttk.Frame(tab\_control)

tab\_control.add(self.ticket\_frame, text="Ticket Purchasing")

self.setup\_ticket\_purchasing()

# Admin Dashboard Tab

self.admin\_frame = ttk.Frame(tab\_control)

tab\_control.add(self.admin\_frame, text="Admin Dashboard")

self.setup\_admin\_dashboard()

tab\_control.pack(expand=1, fill="both")

def setup\_account\_management(self):

tk.Label(self.account\_frame, text="Login Page", font=("Arial", 16)).pack(pady=10)

# Login Form

tk.Label(self.account\_frame, text="Username").pack(pady=5)

self.username\_entry = tk.Entry(self.account\_frame)

self.username\_entry.pack(pady=5)

tk.Label(self.account\_frame, text="Password").pack(pady=5)

self.password\_entry = tk.Entry(self.account\_frame, show="\*")

self.password\_entry.pack(pady=5)

tk.Button(self.account\_frame, text="Login", command=self.login\_account).pack(pady=10)

# Placeholder for account actions

self.account\_actions\_frame = tk.Frame(self.account\_frame)

self.account\_actions\_frame.pack(pady=20)

def login\_account(self):

username = self.username\_entry.get()

password = self.password\_entry.get()

# Automatically create the account if it doesn't exist

if username not in self.accounts:

self.accounts[username] = {

"password": password,

"email": f"{username}@example.com",

"phone": "+0000000000", # Default phone number

}

self.purchase\_orders[username] = [] # Initialize empty purchase orders

if username in self.accounts and self.accounts[username]["password"] == password:

messagebox.showinfo("Login", "Login successful!")

self.display\_account\_actions(username)

else:

messagebox.showerror("Login", "Invalid username or password!")

def display\_account\_actions(self, username):

# Clear previous actions

for widget in self.account\_actions\_frame.winfo\_children():

widget.destroy()

tk.Label(self.account\_actions\_frame, text=f"Welcome, {username}", font=("Arial", 14)).pack(pady=10)

# Display Customer Details

details = self.accounts[username]

details\_text = f"Email: {details['email']}\nPhone: {details['phone']}"

tk.Label(self.account\_actions\_frame, text=details\_text).pack(pady=10)

# Purchase Order Management

tk.Button(self.account\_actions\_frame, text="View Purchase Orders", command=lambda: self.view\_purchase\_orders(username)).pack(pady=5)

tk.Button(self.account\_actions\_frame, text="Modify Purchase Orders", command=lambda: self.modify\_purchase\_orders(username)).pack(pady=5)

tk.Button(self.account\_actions\_frame, text="Delete Purchase Orders", command=lambda: self.delete\_purchase\_orders(username)).pack(pady=5)

def view\_purchase\_orders(self, username):

view\_window = tk.Toplevel(self.root)

view\_window.title("View Purchase Orders")

view\_window.geometry("400x300")

orders = self.purchase\_orders[username]

if not orders:

tk.Label(view\_window, text="No purchase orders found.", font=("Arial", 12)).pack(pady=10)

else:

for order in orders:

tk.Label(view\_window, text=f"Order ID: {order['id']} - Ticket: {order['ticket']} - Quantity: {order['quantity']}", font=("Arial", 10)).pack(pady=5)

def modify\_purchase\_orders(self, username):

modify\_window = tk.Toplevel(self.root)

modify\_window.title("Modify Purchase Orders")

modify\_window.geometry("400x300")

orders = self.purchase\_orders[username]

if not orders:

tk.Label(modify\_window, text="No purchase orders to modify.", font=("Arial", 12)).pack(pady=10)

else:

tk.Label(modify\_window, text="Select an Order to Modify:", font=("Arial", 12)).pack(pady=10)

order\_ids = [order['id'] for order in orders]

selected\_order = tk.StringVar(value=order\_ids[0])

# Dropdown for selecting an order

order\_menu = ttk.Combobox(modify\_window, textvariable=selected\_order, values=order\_ids)

order\_menu.pack(pady=10)

# Fields to modify

tk.Label(modify\_window, text="New Ticket Type").pack(pady=5)

new\_ticket\_entry = tk.Entry(modify\_window)

new\_ticket\_entry.pack(pady=5)

tk.Label(modify\_window, text="New Quantity").pack(pady=5)

new\_quantity\_entry = tk.Entry(modify\_window)

new\_quantity\_entry.pack(pady=5)

tk.Button(modify\_window, text="Save Changes", command=lambda: self.save\_order\_changes(username, selected\_order.get(), new\_ticket\_entry.get(), new\_quantity\_entry.get(), modify\_window)).pack(pady=10)

def save\_order\_changes(self, username, order\_id, new\_ticket, new\_quantity, modify\_window):

for order in self.purchase\_orders[username]:

if order['id'] == order\_id:

order['ticket'] = new\_ticket

order['quantity'] = new\_quantity

messagebox.showinfo("Modify Order", "Order details updated successfully!")

break

modify\_window.destroy()

def delete\_purchase\_orders(self, username):

delete\_window = tk.Toplevel(self.root)

delete\_window.title("Delete Purchase Orders")

delete\_window.geometry("400x300")

orders = self.purchase\_orders[username]

if not orders:

tk.Label(delete\_window, text="No purchase orders to delete.", font=("Arial", 12)).pack(pady=10)

else:

tk.Label(delete\_window, text="Select an Order to Delete:", font=("Arial", 12)).pack(pady=10)

order\_ids = [order['id'] for order in orders]

selected\_order = tk.StringVar(value=order\_ids[0])

# Dropdown for selecting an order

order\_menu = ttk.Combobox(delete\_window, textvariable=selected\_order, values=order\_ids)

order\_menu.pack(pady=10)

tk.Button(delete\_window, text="Delete Order", command=lambda: self.remove\_order(username, selected\_order.get(), delete\_window)).pack(pady=10)

def remove\_order(self, username, order\_id, delete\_window):

self.purchase\_orders[username] = [order for order in self.purchase\_orders[username] if order['id'] != order\_id]

messagebox.showinfo("Delete Order", "Order deleted successfully!")

delete\_window.destroy()

# Ticket Purchasing Section

def setup\_ticket\_purchasing(self):

tk.Label(self.ticket\_frame, text="Ticket Purchasing Interface", font=("Arial", 16)).pack(pady=10)

# Ticket Options

tickets = [

{"Type": "Single-Day Pass", "Price": "275 DHS", "Validity": "1 Day"},

{"Type": "Two-Day Pass", "Price": "480 DHS", "Validity": "2 Days"},

{"Type": "Annual Membership", "Price": "1840 DHS", "Validity": "1 Year"},

{"Type": "Child Ticket", "Price": "185 DHS", "Validity": "1 Day"},

{"Type": "Group Ticket", "Price": "220 DHS", "Validity": "1 Day"},

]

ticket\_table = ttk.Treeview(self.ticket\_frame, columns=("Type", "Price", "Validity"), show="headings")

ticket\_table.heading("Type", text="Type")

ticket\_table.heading("Price", text="Price")

ticket\_table.heading("Validity", text="Validity")

ticket\_table.pack(pady=10)

for ticket in tickets:

ticket\_table.insert("", tk.END, values=(ticket["Type"], ticket["Price"], ticket["Validity"]))

tk.Button(self.ticket\_frame, text="Proceed to Payment", command=self.open\_payment\_interface).pack(pady=10)

def open\_payment\_interface(self):

payment\_window = tk.Toplevel(self.root)

payment\_window.title("Payment Interface")

payment\_window.geometry("400x400")

tk.Label(payment\_window, text="Select Payment Method", font=("Arial", 14)).pack(pady=10)

# Payment Options

payment\_method = tk.StringVar(value="Credit Card")

tk.Radiobutton(payment\_window, text="Credit Card", variable=payment\_method, value="Credit Card").pack(pady=5)

tk.Radiobutton(payment\_window, text="Debit Card", variable=payment\_method, value="Debit Card").pack(pady=5)

tk.Label(payment\_window, text="Card Number").pack(pady=5)

tk.Entry(payment\_window, width=30).pack(pady=5)

tk.Label(payment\_window, text="Cardholder Name").pack(pady=5)

tk.Entry(payment\_window, width=30).pack(pady=5)

tk.Label(payment\_window, text="Card CVV").pack(pady=5)

tk.Entry(payment\_window, width=10, show="\*").pack(pady=5)

tk.Button(payment\_window, text="Pay Now", command=lambda: self.process\_payment(payment\_method.get())).pack(pady=20)

def process\_payment(self, method):

messagebox.showinfo("Payment", f"Payment successful using {method}!")

# Admin Dashboard with Ticket Sales and Discount Modifications

def setup\_admin\_dashboard(self):

tk.Label(self.admin\_frame, text="Admin Dashboard", font=("Arial", 16)).pack(pady=10)

# Display Ticket Sales

tk.Label(self.admin\_frame, text="Ticket Sales (Number Sold per Day)").pack(pady=10)

sales\_data = {

"2024-12-01": 150,

"2024-12-02": 200,

"2024-12-03": 175,

}

sales\_table = ttk.Treeview(self.admin\_frame, columns=("Date", "Tickets Sold"), show="headings")

sales\_table.heading("Date", text="Date")

sales\_table.heading("Tickets Sold", text="Tickets Sold")

sales\_table.pack(pady=10)

for date, sold in sales\_data.items():

sales\_table.insert("", tk.END, values=(date, sold))

# Modify Discounts

tk.Button(self.admin\_frame, text="Modify Discounts", command=self.modify\_discounts).pack(pady=10)

def modify\_discounts(self):

discount\_window = tk.Toplevel(self.root)

discount\_window.title("Modify Discounts")

discount\_window.geometry("300x200")

tk.Label(discount\_window, text="Set New Discount for Tickets").pack(pady=10)

tk.Entry(discount\_window, width=20).pack(pady=5) # Discount percentage input

tk.Button(discount\_window, text="Update Discount", command=lambda: messagebox.showinfo("Discount", "Discount updated!")).pack(pady=10)

# Main Execution

if \_name\_ == "\_main\_":

root = tk.Tk()

app = ThemeParkGUI(root)

 root.mainloop()

**Test Case Output:**

A screenshot of a computer

Description automatically generated

**GUI TEST OUTPUT:**

This picture represents the login page for the Booking system for booking tickets for the theme park. You enter the username and password and then you can view the personalized account details and manage purchase orders.

A screenshot of a login page

Description automatically generated

This photo represents the page for purchasing tickets which displays the available ticket types, their prices, and the validity details with and option to proceed to payment.

A screenshot of a computer

Description automatically generated

This picture represents the payment interface which allows users to select the two types of payment method (credit, debit) and input the card details such as the name, card number, and the CVV.

A screenshot of a computer

Description automatically generated

This picture represents the Admin Dashboard which shows the number of tickets sold per day with an option to modify discount rates for the tickets.

A screenshot of a computer

Description automatically generated

This picture represents the discount modification interface which allows administrators to set and update new discounted rates for tickets.

A screenshot of a computer

Description automatically generated

**Github link:**