Final Project (Assignment 3)

ICS220 > 23018 Program. Fund.

Prof. Areej Abdulfattah

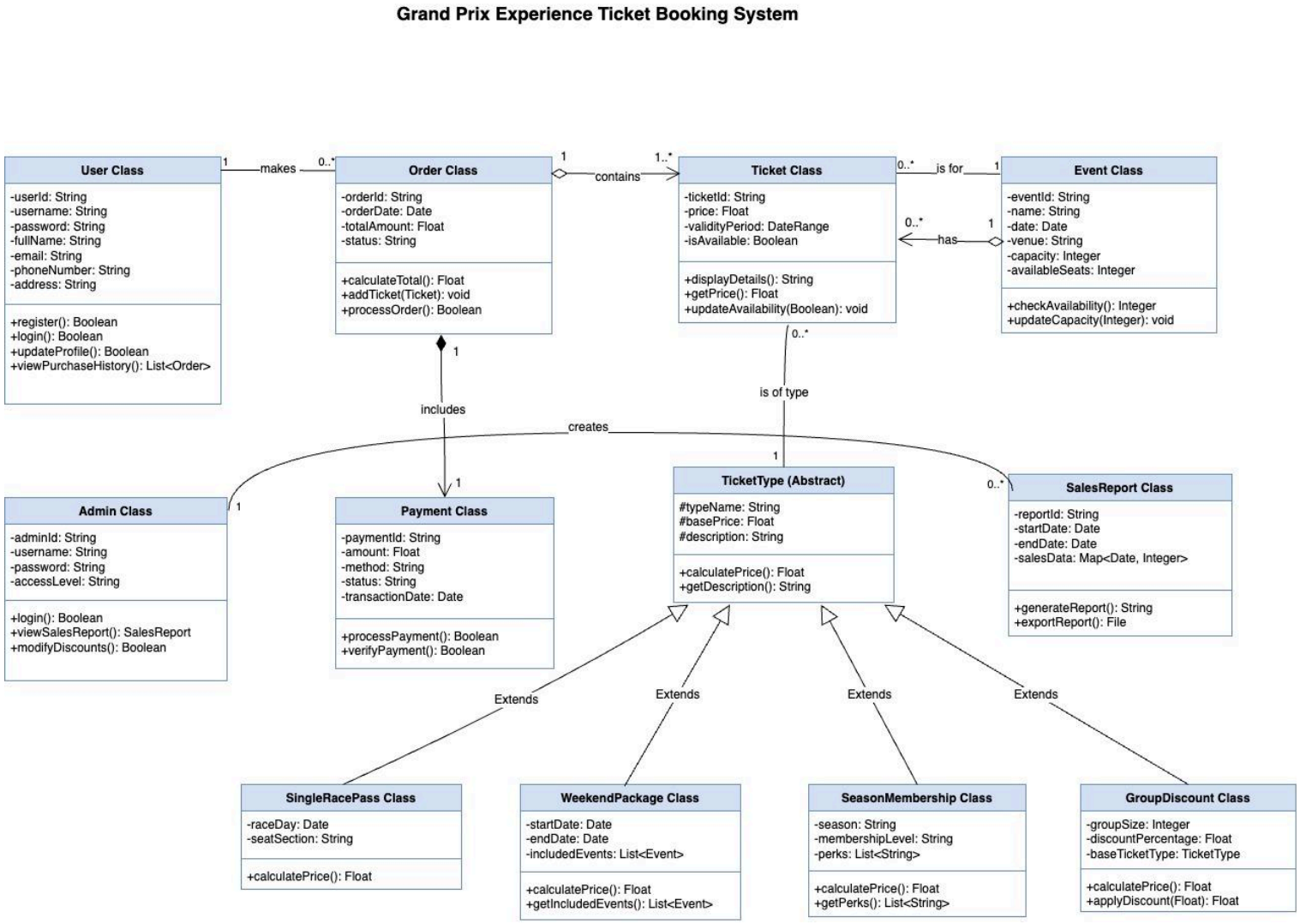
Salma Almansoori - 202317014

Hessa Alzaabi - 202303926

13 May 2025

UML Class Diagram

This UML Class Diagram shows how our Grand Prix Ticket Booking System works. We're creating a system that helps racing fans buy tickets for Formula One events. The diagram shows all the important parts of our system and how they connect to each other. Our ticket booking system lets racing fans create accounts, buy different types of tickets, and manage their bookings. The system also has tools for admins to track sales and manage discounts, which it handles four types of tickets: single-race passes, weekend packages, season memberships, and group discounts.



Class Structure

User Class

- Purpose: Represents customers who register and use the ticketing system.
- Attributes:
    - -userId: String: Unique identifier for each user
    - -username: String: User's login name
    - -password: String: User's password (securely stored)
    - -fullName: String: User's complete name
    - -email: String: User's email address
    - -phoneNumber: String: User's contact number
    - -address: String: User's physical address
- Methods:
    - +register(): Boolean: Registers a new user in the system
    - +login(): Boolean: Authenticates user credentials
    - +updateProfile(): Boolean: Updates user information
    - +viewPurchaseHistory(): List<Order>: Retrieves user's past orders

Ticket Class

- Purpose: Represents an individual ticket that can be purchased.
- Attributes:
    - -ticketId: String: Unique identifier for each ticket
    - -price: Float: Cost of the ticket
    - -validityPeriod: DateRange: Period when the ticket is valid
    - -isAvailable: Boolean: Whether the ticket is available for purchase
- Methods:
    - +displayDetails(): String: Shows information about the ticket
    - +getPrice(): Float: Returns the price of the ticket
    - +updateAvailability(Boolean): void: Updates ticket availability

TicketType Class (Abstract)

- Purpose: Base class for different categories of tickets.
- Attributes:
    - #typeName: String: Name of the ticket type
    - #basePrice: Float: Starting price for the ticket type
    - #description: String: Description of what the ticket includes
- Methods:
    - +calculatePrice(): Float: Abstract method to calculate ticket price

    - +getDescription(): String: Returns ticket type description

Event Class

- Purpose: Represents a specific racing event.
- Attributes:
    - -eventId: String: Unique identifier for each event
    - -name: String: Name of the racing event
    - -date: Date: When the event takes place
    - -venue: String: Location of the event
    - -capacity: Integer: Maximum number of attendees
    - -availableSeats: Integer: Current number of available seats
- Methods:
    - +checkAvailability(): Integer: Checks how many seats are available
    - +updateCapacity(Integer): void: Updates available capacity

Order Class

- Purpose: Represents a purchase transaction.
- Attributes:
    - -orderId: String: Unique identifier for each order
    - -orderDate: Date: When the order was placed
    - -totalAmount: Float: Total cost of the order
    - -status: String: Current status of the order (e.g., pending, confirmed)
- Methods:
    - +calculateTotal(): Float: Calculates the total price
    - +addTicket(Ticket): void: Adds a ticket to the order
    - +processOrder(): Boolean: Processes the complete order

Payment Class

- Purpose: Handles payment processing for orders.
- Attributes:
    - -paymentId: String: Unique identifier for each payment
    - -amount: Float: Amount paid
    - -method: String: Payment method used
    - -status: String: Status of the payment
    - -transactionDate: Date: When the payment was processed
- Methods:
    - +processPayment(): Boolean: Processes the payment
    - +verifyPayment(): Boolean: Verifies payment success

Admin Class

- Purpose: Represents system administrators.
- Attributes:
    - -adminId: String: Unique identifier for each admin
    - -username: String: Admin's login name
    - -password: String: Admin's password
    - -accessLevel: String: Level of system access
- Methods:
    - +login(): Boolean: Authenticates admin credentials
    - +viewSalesReport(): SalesReport: Generates sales reports
    - +modifyDiscounts(): Boolean: Manages discount availability

SalesReport Class

- Purpose: Generates reports for administrators.
- Attributes:
    - -reportId: String: Unique identifier for each report
    - -startDate: Date: Beginning date for the report
    - -endDate: Date: Ending date for the report
    - -salesData: Map<Date, Integer>: Daily sales figures
- Methods:
    - +generateReport(): String: Creates a formatted report
    - +exportReport(): File: Exports report to a file

Specialized Ticket Type Classes

SingleRacePass Class (extends TicketType)

- Attributes:
    - -raceDay: Date: The specific race day
    - -seatSection: String: Seating section information
- Methods:
    - +calculatePrice(): Float: Overrides base method to calculate price

WeekendPackage Class (extends TicketType)

- Attributes:
    - -startDate: Date: Beginning of the package
    - -endDate: Date: End of the package
    - -includedEvents: List<Event>: Events included in the package

- Methods:
    - +calculatePrice(): Float: Overrides base method to calculate price
    - +getIncludedEvents(): List<Event>: Returns included events

SeasonMembership Class (extends TicketType)

- Attributes:
    - -season: String: The racing season covered
    - -membershipLevel: String: Level of membership
    - -perks: List<String>: Additional benefits included
- Methods:
    - +calculatePrice(): Float: Overrides base method to calculate price
    - +getPerks(): List<String>: Returns membership perks

GroupDiscount Class (extends TicketType)

- Attributes:
    - -groupSize: Integer: Minimum number of people for discount
    - -discountPercentage: Float: Percentage of discount applied
    - -baseTicketType: TicketType: The ticket type to which discount applies
- Methods:
    - +calculatePrice(): Float: Overrides base method to calculate price
    - +applyDiscount(Float): Float: Applies the discount to a price

<u>Relationships Between Classes</u>

Association Relationships

1. **User and Order**
- Description: A user can make many orders over time
- Multiplicity: One User can have Many Orders
- The User keeps track of their orders, and each Order knows which User made it
2. **Ticket and Event**
- Description: Each ticket is for a specific event
- Multiplicity: Many Tickets for One Event
- Each Ticket has information about which Event it's for
3. **Admin and Sales Report**
- Description: Admins can create and view reports about ticket sales
- Multiplicity: One Admin can make Many Reports
- Admins have tools to generate reports whenever needed

Aggregation Relationships

1. **Order and Ticket**

- Description:An order contains multiple tickets, but tickets can exist without being in an order
- Multiplicity: One Order can have Many Tickets
- The Order keeps a list of which Tickets are included

2. **Event and Ticket**

- Description: An event has many tickets available, but tickets can exist before being assigned to events
- Multiplicity: One Event has Many Tickets
- The Event keeps track of how many tickets are available

Composition Relationship

1. **Order and Payment**

- Description: A payment is created as part of an order and can't exist without it
- Multiplicity: One Order has exactly One Payment
- When an Order is created, it creates its own Payment

Inheritance Relationships

1. **TicketType and Special Ticket Types**

- Description: The main TicketType class is like a parent to four specialized ticket types
- Multiplicity: Each child class gets features from TicketType but adds its own special features
- The children: SingleRacePass, WeekendPackage, SeasonMembership, and GroupDiscount
- Each child calculates prices differently based on its type

Assumptions Made

1. User Authentication: Users must register and log in before purchasing tickets.
2. Ticket Availability: Each event has a limited capacity, and ticket availability is tracked in real-time.
3. Payment Processing: The system supports multiple payment methods, with payment processing handled securely.
4. Discount Application: Different types of discounts can be applied to tickets based on specific rules.
5. Order States: Orders progress through various states (pending, confirmed, completed, canceled).

6. Data Persistence: All system data will be stored in binary files using Python's Pickle library.
7. Ticket Types: Four distinct ticket types are offered, each with its own pricing model.
8. Admin Roles: Administrators have varying access levels that determine their system privileges.

Python Classes

__init__.py:

```
from Models.user import User
from Models.ticket import Ticket
from Models.ticket_type import TicketType, SingleRacePass, WeekendPackage,
SeasonMembership, GroupDiscount
from Models.order import Order
from Models.payment import Payment
from Models.admin import Admin
from Models.sales_report import SalesReport
```

user.py:

```
# Creates a new user with personal information

class User:

   def __init__(self, user_id, username, password, salt, full_name, email,
phone_number, address):

        # Store all user details privately

        self._user_id = user_id

        self._username = username

        self._password = password

        self._salt = salt  # Store salt

        self._full_name = full_name

        self._email = email

        self._phone_number = phone_number

        self._address = address

        self._purchase_history = []  # Keep track of orders
```

```
# Get the user's ID
def get_user_id(self):
    return self._user_id

# Get the username
def get_username(self):
    return self._username

# Get the password
def get_password(self):
    return self._password
#Get the salt used for hashing the user's password
def get_salt(self):
    return self._salt

# Get the full name
def get_full_name(self):
    return self._full_name

# Get the email address
def get_email(self):
    return self._email

# Get the phone number
def get_phone_number(self):
    return self._phone_number
```

```
# Get the address
def get_address(self):
    return self._address

# Get the list of past orders
def get_purchase_history(self):
    return self._purchase_history

# Update the username
def set_username(self, username):
    self._username = username

# Update the password
def set_password(self, password):
    self._password = password

# Update the full name
def set_full_name(self, full_name):
    self._full_name = full_name

# Update the email address
def set_email(self, email):
    self._email = email

# Update the phone number
```

```
    def set_phone_number(self, phone_number):
        self._phone_number = phone_number

    # Update the address
    def set_address(self, address):
        self._address = address

    # Register a new user in the system
    def register(self):
        return True

    # Check if login details are correct
    def login(self, username, password):
        return self._username == username and self._password == password

    # Update user's personal information
    def update_profile(self, full_name=None, email=None, phone_number=None,
address=None):
        if full_name:
            self._full_name = full_name
        if email:
            self._email = email
        if phone_number:
            self._phone_number = phone_number
        if address:
            self._address = address
```

```
    # Add an order to the user's purchase history
    def add_purchase(self, order):
        self._purchase_history.append(order)

    # Show all previous purchases
    def view_purchase_history(self):
        return self._purchase_history
```

ticket.py:

```
# Represents an event ticket
class Ticket:
    def __init__(self, ticket_id, price, validity_start, validity_end,
is_available=True):
        self._ticket_id = ticket_id  # Unique ID for this ticket
        self._price = price  # How much the ticket costs
        self._validity_period = (validity_start, validity_end)  # When ticket is
valid
        self._is_available = is_available  # Availability status

    # Get the ticket ID
    def get_ticket_id(self):
        return self._ticket_id

    # Get the ticket price
    def get_price(self):
        return self._price
```

```
    # Get when this ticket is valid
    def get_validity_period(self):
        return self._validity_period

    # Check if this ticket can be bought
    def is_available(self):
        return self._is_available

    # Change the ticket price
    def set_price(self, price):
        self._price = price

    # Change when the ticket is valid
    def set_validity_period(self, start, end):
        self._validity_period = (start, end)

    # Mark ticket as available or sold
    def update_availability(self, available):
        self._is_available = available

    # Show ticket information
    def display_details(self):
        start, end = self._validity_period
        return (
            f"Ticket ID: {self._ticket_id}, "
```

```
            f"Price: ${self._price:.2f}, "
            f"Validity: {start} to {end}, "
            f"Available: {'Yes' if self._is_available else 'No'}"
        )
    #Get the name of the Ticket type
    def get_type_name(self):
        return "Ticket"

    #Get the ticket event
    def get_event(self):
        return None
```

ticket_type.py:

```
from abc import ABC, abstractmethod
class TicketType(ABC):
  # Base class for different types of tickets
  def __init__(self, type_name, base_price, description=""):
      self._type_name = type_name  # Name of this ticket type
      self._base_price = base_price  # Starting price
      self._description = description  # What this ticket includes

  # Get the name of this ticket type
  def get_type_name(self):
      return self._type_name
```

```
    # Get the base price
    def get_base_price(self):
        return self._base_price

    # Get the description
    def get_description(self):
        return self._description

    # Change the ticket type name
    def set_type_name(self, name):
        self._type_name = name

    # Change the base price
    def set_base_price(self, price):
        self._base_price = price

    # Change the description
    def set_description(self, description):
        self._description = description

    # Calculate the final price
    @abstractmethod
    def calculate_price(self, quantity=1):
        raise NotImplementedError("Must override in subclass")
```

```
class SingleRacePass(TicketType):

  # Ticket for just one race day

  def __init__(self, type_name, base_price, description, race_day,
seat_section):

      super().__init__(type_name, base_price, description)

      self._race_day = race_day  # Which day is the race

      self._seat_section = seat_section  # Where you sit


  # Get the race day

  def get_race_day(self):

      return self._race_day


  # Get the seat section

  def get_seat_section(self):

      return self._seat_section


  # Change the race day

  def set_race_day(self, day):

      self._race_day = day


  # Change the seat section

  def set_seat_section(self, section):

      self._seat_section = section


  # Calculate price for single race

  def calculate_price(self, quantity=1):
```

```
        return (self._base_price + 10) * quantity



class WeekendPackage(TicketType):

    # Ticket for the whole weekend with multiple events

    def __init__(self, type_name, base_price, description, start_date, end_date,
included_events):

        super().__init__(type_name, base_price, description)

        self._start_date = start_date  # Weekend starts

        self._end_date = end_date  # Weekend ends

        self._included_events = included_events  # What events are included


    # Get weekend start date

    def get_start_date(self):

        return self._start_date


    # Get weekend end date

    def get_end_date(self):

        return self._end_date


    # Get list of included events

    def get_included_events(self):

        return self._included_events


    # Calculate price

    def calculate_price(self, quantity=1):
```

```
        return (self._base_price + len(self._included_events) * 20) * quantity



class SeasonMembership(TicketType):
  # Membership for the entire racing season
  def __init__(self, type_name, base_price, description, season,
membership_level, perks):
      super().__init__(type_name, base_price, description)
      self._season = season  # Which season
      self._membership_level = membership_level  # Bronze, Silver, Gold, etc.
      self._perks = perks  # Extra benefits included

  # Get the season year
  def get_season(self):
      return self._season

  # Get membership level
  def get_membership_level(self):
      return self._membership_level

  # Get list of perks
  def get_perks(self):
      return self._perks

  # Calculate price
  def calculate_price(self, quantity=1):
```

```
        return (self._base_price + len(self._perks) * 15) * quantity



class GroupDiscount(TicketType):

    def __init__(self, type_name, base_price, description, group_size,
discount_percentage, base_ticket_type):

        super().__init__(type_name, base_price, description)

        self._group_size = group_size

        self._discount_percentage = discount_percentage

        self._base_ticket_type = base_ticket_type




    def get_group_size(self):

        return self._group_size




    def get_discount_percentage(self):

        return self._discount_percentage




    def get_base_ticket_type(self):

        return self._base_ticket_type




    def calculate_price(self, quantity=1):

        base_price = self._base_ticket_type.calculate_price(quantity)
```

```
        if quantity >= self._group_size:
            return base_price * (1 - self._discount_percentage / 100)
        return base_price


    def set_discount_percentage(self, percentage):  # Added method
        self._discount_percentage = percentage
```

event.py

```
class Event:
    # Creates a new racing event
    def __init__(self, event_id, name, date, venue, capacity):
        self._event_id = event_id  # Unique ID for this event
        self._name = name  # Name of the event
        self._date = date  # When the event happens
        self._venue = venue  # Where the event takes place
        self._capacity = capacity  # Maximum number of people
        self._available_seats = capacity  # How many seats are left

    # Get the event ID
    def get_event_id(self):
        return self._event_id

    # Get the event name
    def get_name(self):
        return self._name
```

```
# Get the event date
def get_date(self):
    return self._date

# Get the venue location
def get_venue(self):
    return self._venue

# Get total capacity
def get_capacity(self):
    return self._capacity

# Get available seats left
def get_available_seats(self):
    return self._available_seats

# Change the event name
def set_name(self, name):
    self._name = name

# Change the event date
def set_date(self, date):
    self._date = date

# Change the venue
```

```
def set_venue(self, venue):
    self._venue = venue

# Change total capacity
def set_capacity(self, capacity):
 if capacity < 0:
     raise ValueError("Capacity must be non-negative")
 self._capacity = capacity
 if self._available_seats > capacity:
     self._available_seats = capacity  # Adjust available seats down

# Update available seats
def set_available_seats(self, seats):
 if not 0 <= seats <= self._capacity:
     raise ValueError("Available seats must be between 0 and capacity")
 self._available_seats = seats

# Check how many seats are still available
def check_availability(self):
    return self._available_seats

# Update the number of available seats
def reduce_seats(self, quantity):
    if quantity > self._available_seats:
        raise ValueError("Not enough seats available")
    self._available_seats -= quantity
```

```
    def increase_seats(self, quantity):
        if self._available_seats + quantity > self._capacity:
            raise ValueError("Exceeds maximum capacity")
        self._available_seats += quantity
```

order.py:

```
class Order:
    # Creates a new order for ticket purchases
    def __init__(self, order_id, order_date, status="pending", user=None):
        self._order_id = order_id        # Unique ID for this order
        self._order_date = order_date    # When the order was made
        self._tickets = []               # List of tickets in this order
        self._status = status            # Current order status
        self._user = user                # Who placed this order

    # Get the order ID
    def get_order_id(self):
        return self._order_id

    # Get when the order was placed
    def get_order_date(self):
        return self._order_date

    # Get all tickets in this order
    def get_tickets(self):
```

```
    return self._tickets

def get_user_id(self):
     return self._user.get_user_id() if self._user else None

def get_total(self):
     return sum(ticket.get_price() for ticket in self._tickets)

# Get the order status
def get_status(self):
    return self._status

# Get who made this order
def get_user(self):
    return self._user

def get_ticket_type(self):
     return self._tickets[0].get_type_name() if self._tickets else "Unknown"

# Change the order status
def set_status(self, status):
    if status not in ("pending", "confirmed", "cancelled"):
        raise ValueError("Invalid order status")
    self._status = status

# Add a ticket to this order
```

```
  def add_ticket(self, ticket):
      if hasattr(ticket, "get_price") and callable(ticket.get_price):
          self._tickets.append(ticket)
      else:
          raise TypeError("Invalid ticket object: missing get_price method")

  # Remove a ticket from the order
  def remove_ticket(self, ticket):
      if ticket in self._tickets:
          self._tickets.remove(ticket)

      # Confirm order and reduce available seats for each ticket's event
      def process_order(self):
          if not self._tickets:
              raise ValueError("Cannot process an empty order")

          for ticket in self._tickets:
              if hasattr(ticket, "get_event") and ticket.get_event():
                  event = ticket.get_event()
                  if hasattr(event, "reduce_seats") and
callable(event.reduce_seats):
                      event.reduce_seats(1)
                  else:
                      raise AttributeError("Event does not support seat
reduction")
              else:
```

```
                raise ValueError("Ticket is not associated with a valid
event")


        self._status = "confirmed"

        return True
```

payment.py

```
from datetime import datetime

class Payment:

   #Allows multiple payment methods

   ACCEPTED_METHODS = ["credit_card", "debit_card", "cash", "paypal"]



   # Handles payment for an order

   def __init__(self, payment_id, amount, method, status="pending",
transaction_date=None):

       if method not in self.ACCEPTED_METHODS:

           raise ValueError(f"Unsupported payment method. Choose from: {',
'.join(self.ACCEPTED_METHODS)}")

       self._payment_id = payment_id

       self._amount = amount

       self._method = method

       self._status = status

       self._transaction_date = transaction_date



   # Get the payment ID

   def get_payment_id(self):

      return self._payment_id
```

```
# Get the payment amount
def get_amount(self):
   return self._amount

# Get the payment method
def get_method(self):
   return self._method

# Get the payment status
def get_status(self):
   return self._status

# Get when payment was made
def get_transaction_date(self):
   return self._transaction_date

# Change the payment status
def set_status(self, status):
   self._status = status

#Change the transaction date
def set_transaction_date(self, date=None):
    self._transaction_date = datetime.now()

# Process the payment
```

```
    def process_payment(self):
        self._status = "processed"
        if self._transaction_date is None:
            self._transaction_date = datetime.now().strftime("%Y-%m-%d
%H:%M:%S")
        return True

    # Verify if payment was successful based on the method
    def verify_payment(self):
        if self._method in ["credit_card", "debit_card", "paypal"]:
            return self._status == "processed"
        elif self._method == "cash":
            return self._status == "paid_on_delivery"
        return False
```

admin.py

```
from .sales_report import SalesReport
from .ticket_type import TicketType

class Admin:
  # Creates an admin user for managing the system
  def __init__(self, admin_id, username, password, access_level="full"):
      self._admin_id = admin_id  # Unique admin ID
      self._username = username  # Admin login name
      self._password = password  # Admin password
```

```
        self._access_level = access_level  # What admin can access

    # Get the admin ID
    def get_admin_id(self):
        return self._admin_id

    # Get the admin username
    def get_username(self):
        return self._username

    # Get the admin password
    def get_password(self):
        return self._password

    # Get the access level
    def get_access_level(self):
        return self._access_level

    # Change the access level
    def set_access_level(self, level):
        self._access_level = level

    # Admin login to system
    def login(self, username, password):
        return self._username == username and self._password == password #Checks
admin username and password
```

```
    # Create and view sales reports
    def view_sales_report(self, report: SalesReport):
        return report.generate_report()

    # Apply and modify discount settings
    def apply_discount(self, ticket_type: TicketType, percentage: float):
        if not 0 <= percentage <= 100:
            raise ValueError("Discount must be between 0 and 100%")
        ticket_type.apply_discount(percentage)
```

sales_report.py

```
import datetime
class SalesReport:
    # Creates reports about ticket sales
    def __init__(self, report_id, start_date, end_date, sales_data=None):
        self._report_id = report_id  # Unique report ID
        self._start_date = start_date  # Report period starts
        self._end_date = end_date  # Report period ends
        # Initialize sales data as an empty dictionary if not passed
        self._sales_data = sales_data if sales_data is not None else {}

    # Get the report ID
    def get_report_id(self):
        return self._report_id
```

```
  # Get report start date
  def get_start_date(self):
      return self._start_date

  # Get report end date
  def get_end_date(self):
      return self._end_date

  # Get sales data
  def get_sales_data(self):
      return self._sales_data

  # Record a ticket sale on a given date
  def record_sale(self, sale_date):
       if isinstance(sale_date, datetime.date):
           sale_date = sale_date
           if sale_date not in self._sales_data:
               self._sales_data[sale_date] = 0
           self._sales_data[sale_date] += 1

  # Generate a formatted report
  def generate_report(self):
      header = f"Sales Report: {self._report_id} from {self._start_date} to
{self._end_date}"
      report_lines = [f"{date}: {count} ticket(s) sold" for date, count in
sorted(self._sales_data.items())]
      return header + "\n" + "\n".join(report_lines)
```

```
    # Save report to a file
    def export_report(self):
        filename = f"{self._report_id}_sales_report.txt"
        with open(filename, "w") as file:
            file.write(self.generate_report())
        return filename
```

TicketBookingSystem.py:

```
# Import all needed libraries
import tkinter as tk
from tkinter import ttk, messagebox
from datetime import date, datetime
import pickle
import hashlib
import os

# Import our custom classes
from Models.user import User
from Models.ticket import Ticket
from Models.ticket_type import TicketType, SingleRacePass, WeekendPackage,
SeasonMembership, GroupDiscount
from Models.order import Order
from Models.payment import Payment
from Models.admin import Admin
from Models.sales_report import SalesReport
```

```
# Files where we save data
USERS_FILE = "users.pickle"
ORDERS_FILE = "orders.pickle"

# Helper functions for file operations
def load_data(filename):
  """Load data from a pickle file"""
  try:
      with open(filename, "rb") as f:
          return pickle.load(f)
  except (FileNotFoundError, EOFError):
      return []

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

def hash_password(password):
  """Make password secure by hashing it"""
  salt = os.urandom(16)
  hashed = hashlib.sha256(salt + password.encode()).hexdigest()
  return salt, hashed

def verify_password(stored_salt, stored_hash, password):
```

```
    """Verify password using stored salt and hash"""
    return stored_hash == hashlib.sha256(stored_salt +
password.encode()).hexdigest()


# Create different types of tickets
def get_ticket_types():
    race_pass = SingleRacePass("Single Race", 120, "Access to one race day",
"Friday", "Main Grandstand")
    weekend = WeekendPackage("Weekend Pass", 300, "All weekend events",
"2025-11-10", "2025-11-12", ["Practice", "Qualifying", "Race"])
    season = SeasonMembership("Season Ticket", 1000, "All-season access",
"2025", "Gold", ["VIP Lounge", "Pit Access", "Free Merch"])
    group = GroupDiscount("Group Deal", 0, "Discounted for groups", 5, 15,
                          SingleRacePass("Group Race", 120, "Group race day",
"Friday", "South Zone"))

    return {
        "RacePass": race_pass,
        "Weekend": weekend,
        "Season": season,
        "Group": group
    }


# Create the ticket types once at startup
TICKET_TYPES = get_ticket_types()


# Main application class
```

```
class BookingSystemApp:
  def __init__(self, root):
      """Set up the main application window"""
      self.root = root
      self.root.title("Grand Prix Ticket Booking System")
      self.root.geometry("600x500")
      self.users = load_data(USERS_FILE)  # Initialize user list here
      self.current_user = None  # Track who is logged in

      # Create tabs for different functions
      self.tabs = ttk.Notebook(self.root)
      self.login_tab()      # Tab for login/register
      self.booking_tab()    # Tab for booking tickets
      self.profile_tab()    # Tab for managing profile
      self.admin_tab()      # Tab for admin functions
      self.tabs.pack(expand=True, fill="both")

      # Disable other tabs until user logs in
      for i in range(1, 4):
          self.tabs.tab(i, state="disabled")

  def login_tab(self):
      """Create the login and registration tab"""
      tab = ttk.Frame(self.tabs)
      self.tabs.add(tab, text="Login / Register")
```

```
        # Create input form
        frame = ttk.LabelFrame(tab, text="Account Access", padding=10)
        frame.pack(padx=20, pady=20)

        # Add labels and input fields
        ttk.Label(frame, text="Username:").grid(row=0, column=0)
        ttk.Label(frame, text="Email:").grid(row=1, column=0)
        ttk.Label(frame, text="Password:").grid(row=2, column=0)

        self.username_entry = ttk.Entry(frame)
        self.email_entry = ttk.Entry(frame)
        self.password_entry = ttk.Entry(frame, show="*")  # Hide password
        self.username_entry.grid(row=0, column=1)
        self.email_entry.grid(row=1, column=1)
        self.password_entry.grid(row=2, column=1)

        # Add buttons
        ttk.Button(frame, text="Register", command=self.register).grid(row=3,
column=0, pady=10)
        ttk.Button(frame, text="Login", command=self.login).grid(row=3,
column=1)

    def booking_tab(self):
        """Create the ticket booking tab"""
        tab = ttk.Frame(self.tabs)
        self.tabs.add(tab, text="Book Ticket")
```

```
        # Create booking form
        frame = ttk.LabelFrame(tab, text="Choose Ticket", padding=10)
        frame.pack(padx=20, pady=20)

        # Ticket type selection
        ttk.Label(frame, text="Ticket Type:").grid(row=0, column=0, sticky="e")
        self.ticket_var = tk.StringVar()
        ticket_menu = ttk.OptionMenu(frame, self.ticket_var, "RacePass",
*TICKET_TYPES.keys())
        ticket_menu.grid(row=0, column=1)

        self.ticket_info_label = ttk.Label(frame, text="", wraplength=400)
        self.ticket_info_label.grid(row=1, column=0, columnspan=2, pady=5)

        self.ticket_var.trace_add("write", self.update_ticket_info)
        self.update_ticket_info()

        # Quantity selection
        ttk.Label(frame, text="Quantity:").grid(row=2, column=0)
        self.qty_entry = ttk.Entry(frame)
        self.qty_entry.grid(row=2, column=1)

        ttk.Label(frame, text="Card Number:").grid(row=3, column=0)
        self.card_entry = ttk.Entry(frame)
        self.card_entry.grid(row=3, column=1)
```

```
        # Book button
        ttk.Button(frame, text="Book Now", command=self.book_ticket).grid(row=4,
columnspan=2, pady=10)



    def profile_tab(self):
        """Create the profile management tab"""
        tab = ttk.Frame(self.tabs)
        self.tabs.add(tab, text="Manage Profile")

        # Create profile form
        frame = ttk.LabelFrame(tab, text="Your Details", padding=10)
        frame.pack(padx=20, pady=10)

        # Profile fields
        ttk.Label(frame, text="Full Name:").grid(row=0, column=0)
        ttk.Label(frame, text="Phone:").grid(row=1, column=0)
        ttk.Label(frame, text="Address:").grid(row=2, column=0)

        self.fullname_entry = ttk.Entry(frame)
        self.phone_entry = ttk.Entry(frame)
        self.address_entry = ttk.Entry(frame)

        self.fullname_entry.grid(row=0, column=1)
        self.phone_entry.grid(row=1, column=1)
        self.address_entry.grid(row=2, column=1)
```

```
        # Update button
        ttk.Button(frame, text="Update Profile",
command=self.update_profile).grid(row=3, columnspan=2, pady=5)

        #Delete account button
        ttk.Button(frame, text="Delete Account",
command=self.delete_account).grid(row=4, columnspan=2, pady=5)

        # Order management
        order_frame = ttk.LabelFrame(tab, text="Your Orders", padding=10)
        order_frame.pack(fill="x", padx=20)

        self.orders_list = tk.Listbox(order_frame, height=5)
        self.orders_list.pack(fill="x", pady=5)

        #Refresh orders button
        ttk.Button(order_frame, text="Refresh Orders",
command=self.load_user_orders).pack()

        #Delete the selected order button
        ttk.Button(order_frame, text="Delete Selected Order",
command=self.delete_selected_order).pack(pady=5)

    def admin_tab(self):
        """Create the admin management tab"""
        tab = ttk.Frame(self.tabs)
        self.tabs.add(tab, text="Admin Panel")
```

```
        # Sales report button
        ttk.Button(tab, text="Generate Sales Report",
command=self.generate_report).pack(pady=10)

        #Modify and apply discount
        ttk.Label(tab, text="Update Group Discount (%):").pack()
        self.discount_entry = ttk.Entry(tab)
        self.discount_entry.pack()
        ttk.Button(tab, text="Apply Discount",
command=self.update_discount).pack(pady=5)

    def show_user_menu(self):
         for i in range(1, 4):
             self.tabs.tab(i, state="normal")

    def register(self):
     username = self.username_entry.get()
     email = self.email_entry.get()
     password = self.password_entry.get()
     if not username or not email or not password:
         messagebox.showerror("Input Error", "Please fill all fields.")
         return
     users = load_data(USERS_FILE)
     if any(u.get_email() == email for u in users):
         messagebox.showerror("Error", "Email already registered.")
         return
```

```
    salt, hashed_password = hash_password(password)
    user_id = f"U{int(datetime.now().timestamp())}"
    user = User(user_id, username, hashed_password, salt, "", email, "", "")
    users.append(user)
    save_data(USERS_FILE, users)
    self.current_user = user
    self.users = users  # Update in-memory list
    self.show_user_menu()
    messagebox.showinfo("Welcome", f"Welcome, {username}! You have been
registered and logged in.")


  def login(self):
        """Log in an existing user"""
        email = self.email_entry.get().lower()
        password = self.password_entry.get()
        for user in self.users:
            print(f"Checking user: {user.get_email().lower()}")
            if user.get_email().lower() == email:
                print("Email match found.")
                user_salt = user.get_salt()
                user_password = user.get_password()
                if user_salt and user_password:
                    if verify_password(user_salt, user_password, password):
                        self.current_user = user
                        messagebox.showinfo("Success", f"Welcome,
{user.get_full_name()}!")
                        self.show_user_menu()
```

```
                    return
                else:
                    messagebox.showerror("Error", "Incorrect password.")
                    return
            else:
                messagebox.showerror("Error", "User data corrupted.")
                return
        messagebox.showerror("Error", "User not found.")

def update_profile(self):
    """Update user profile information"""
    if not self.current_user:
        messagebox.showerror("Not Logged In", "You must log in first.")
        return
    # Get all fields
    username = self.fullname_entry.get().strip()
    phone = self.phone_entry.get().strip()
    address = self.address_entry.get().strip()
    # Validate input
    if not username or not phone or not address:
        messagebox.showerror("Input Error", "Please fill all fields.")
        return
    # Update current user object
    self.current_user.set_full_name(username)  # Corrected method name
    self.current_user.set_phone_number(phone)
    self.current_user.set_address(address)
```

```
        # Update users list
        users = load_data(USERS_FILE)
        for i, user in enumerate(users):
            if user.get_user_id() == self.current_user.get_user_id():
                users[i] = self.current_user
                break
        save_data(USERS_FILE, users)
        messagebox.showinfo("Profile Updated", "Your profile has been
updated.")


  #Delete the user account
  def delete_account(self):
      if not self.current_user:
          return
      confirm = messagebox.askyesno("Confirm", "Delete your account and all
orders?")
      if confirm:
          users = load_data(USERS_FILE)
          users = [u for u in users if u.get_user_id() !=
self.current_user.get_user_id()]
          save_data(USERS_FILE, users)


          orders = load_data(ORDERS_FILE)
          orders = [o for o in orders if o.get_user_id() !=
self.current_user.get_user_id()]
          save_data(ORDERS_FILE, orders)


          self.current_user = None
```

```
            messagebox.showinfo("Deleted", "Account and orders deleted.")
            self.root.destroy()

    #Loads the user orders
    def load_user_orders(self):
     if not self.current_user:
         return
     orders = load_data(ORDERS_FILE)
     self.orders_list.delete(0, tk.END)
     for order in orders:
         if order.get_user().get_user_id() == self.current_user.get_user_id():
             self.orders_list.insert(tk.END, f"{order.get_order_id()} |
{order.get_ticket_type()} x{len(order.get_tickets())} = ${order.get_total()}")

    #Deletes the user selected order
    def delete_selected_order(self):
        if not self.current_user:
            return
        selection = self.orders_list.curselection()
        if not selection:
            messagebox.showerror("Error", "Select an order to delete.")
            return
        order_id = self.orders_list.get(selection[0]).split("|")[0].strip()

        orders = load_data(ORDERS_FILE)
        orders = [o for o in orders if o.get_order_id() != order_id]
```

```
        save_data(ORDERS_FILE, orders)
        self.load_user_orders()
        messagebox.showinfo("Deleted", f"Order {order_id} deleted.")

    #Updates the ticket information
    def update_ticket_info(self, *args):
        ticket = TICKET_TYPES[self.ticket_var.get()]
        info = f"{ticket.get_type_name()}: {ticket.get_description()}\nPrice:
${ticket.calculate_price()}"
        self.ticket_info_label.config(text=info)



    def book_ticket(self):
     """Book tickets for a user"""
     if not self.current_user:
         return
     ticket_type = self.ticket_var.get()
     quantity = self.qty_entry.get()
     card = self.card_entry.get()
     # Validate quantity input
     try:
         quantity = int(quantity)
         if quantity <= 0:
             raise ValueError
     except ValueError:
         messagebox.showerror("Invalid", "Enter a valid quantity.")
```

```
            return
        if len(card) != 16 or not card.isdigit():
            messagebox.showerror("Payment Error", "Enter a 16-digit card number.")
            return
        # Get the selected ticket object
        ticket_obj = TICKET_TYPES[ticket_type]
        total_price = ticket_obj.calculate_price(quantity)
        # Handle group discount dynamically
        if ticket_type == "Group" and quantity >= ticket_obj.get_group_size():
            total_price *= (1 - ticket_obj.get_discount_percentage() / 100)
        # Create payment record
        payment_id = f"P{int(datetime.now().timestamp())}"
        payment = Payment(payment_id, total_price, "credit_card",
transaction_date=datetime.now().strftime("%Y-%m-%d"))
        # Create order
        order_id = f"O{int(datetime.now().timestamp())}"
        order = Order(order_id, datetime.now(), "confirmed", self.current_user)
        # Create and add Ticket objects
        for _ in range(quantity):
            ticket_id = f"T{int(datetime.now().timestamp())}"
            single_ticket_price = ticket_obj.calculate_price(1)  # Price for one
ticket
            ticket = Ticket(ticket_id, single_ticket_price,
datetime.now().strftime("%Y-%m-%d"), datetime.now().strftime("%Y-%m-%d"))
            order.add_ticket(ticket)
        # Save order
        orders = load_data(ORDERS_FILE)
```

```
    orders.append(order)

    save_data(ORDERS_FILE, orders)

    messagebox.showinfo("Booked", f"Tickets booked!\nOrder ID:
{order_id}\nTotal: ${total_price:.2f}")

    self.qty_entry.delete(0, tk.END)

    self.card_entry.delete(0, tk.END)




  def generate_report(self):

        """Generate a sales report"""

        orders = load_data(ORDERS_FILE)

        report_id = f"R{int(datetime.now().timestamp())}"

        current_date = datetime.now().date()

        report = SalesReport(report_id, current_date, current_date)

        for order in orders:

            order_date = order.get_order_date().date() if
isinstance(order.get_order_date(), datetime) else order.get_order_date()

            report.record_sale(order_date)

        total_sales = sum(order.get_total() for order in orders)

        report_text = f"Total orders: {len(orders)}\nTotal Sales:
${total_sales:.2f}\n\n{report.generate_report()}"

        messagebox.showinfo("Sales Report", report_text)



  def update_discount(self):

        """Update the group discount value"""

        new_discount = self.discount_entry.get()

        try:
```

```
            new_discount = float(new_discount)
            if not (0 <= new_discount <= 100):
                raise ValueError
        except ValueError:
            messagebox.showerror("Invalid", "Enter a valid percentage between
0-100.")
            return
        group_ticket = TICKET_TYPES.get("Group")
        if isinstance(group_ticket, GroupDiscount):
            group_ticket.set_discount_percentage(new_discount)  # Corrected
method name
            messagebox.showinfo("Updated", f"Group discount updated to
{new_discount}%.")
        else:
            messagebox.showerror("Error", "Group ticket type not found.")



# --- Launch the app ---
if __name__ == "__main__":
    root = tk.Tk()
    app = BookingSystemApp(root)
    root.mainloop()
```

test_system.py

```
import unittest
from datetime import date
from Models.user import User
from Models.ticket import Ticket
from Models.ticket_type import SingleRacePass, WeekendPackage,
SeasonMembership, GroupDiscount
from Models.event import Event
from Models.order import Order
from Models.payment import Payment
from Models.admin import Admin
from Models.sales_report import SalesReport


class TestUser(unittest.TestCase):

   def setUp(self):
       self.user = User(1, "johndoe", "secure123", "s@lt", "John Doe",
"john@example.com", "123456789", "123 Main St")

   def test_getters(self):
       self.assertEqual(self.user.get_user_id(), 1)
       self.assertEqual(self.user.get_username(), "johndoe")
       self.assertEqual(self.user.get_password(), "secure123")
       self.assertEqual(self.user.get_salt(), "s@lt")
       self.assertEqual(self.user.get_full_name(), "John Doe")
       self.assertEqual(self.user.get_email(), "john@example.com")
       self.assertEqual(self.user.get_phone_number(), "123456789")
       self.assertEqual(self.user.get_address(), "123 Main St")

   def test_setters(self):
       self.user.set_username("janedoe")
       self.assertEqual(self.user.get_username(), "janedoe")

       self.user.set_password("newpass")
       self.assertEqual(self.user.get_password(), "newpass")

       self.user.set_full_name("Jane Doe")
       self.assertEqual(self.user.get_full_name(), "Jane Doe")

       self.user.set_email("jane@example.com")
       self.assertEqual(self.user.get_email(), "jane@example.com")

       self.user.set_phone_number("987654321")
       self.assertEqual(self.user.get_phone_number(), "987654321")
```

```
        self.user.set_address("456 Main St")
        self.assertEqual(self.user.get_address(), "456 Main St")

    def test_login_success(self):
        self.assertTrue(self.user.login("johndoe", "secure123"))

    def test_login_failure(self):
        self.assertFalse(self.user.login("wrong", "wrong"))

    def test_purchase_history(self):
        self.assertEqual(len(self.user.get_purchase_history()), 0)
        self.user.add_purchase("Order1")
        self.assertEqual(len(self.user.get_purchase_history()), 1)
        self.assertIn("Order1", self.user.get_purchase_history())

    def test_update_profile(self):
        self.user.update_profile(full_name="Johnny", email="johnny@example.com",
phone_number="111222333", address="New Place")
        self.assertEqual(self.user.get_full_name(), "Johnny")
        self.assertEqual(self.user.get_email(), "johnny@example.com")
        self.assertEqual(self.user.get_phone_number(), "111222333")
        self.assertEqual(self.user.get_address(), "New Place")
class TestTicket(unittest.TestCase):

    def setUp(self):
        self.ticket = Ticket("T001", 100.0, "2025-01-01", "2025-01-05")

    def test_getters(self):
        self.assertEqual(self.ticket.get_ticket_id(), "T001")
        self.assertEqual(self.ticket.get_price(), 100.0)
        self.assertEqual(self.ticket.get_validity_period(), ("2025-01-01",
"2025-01-05"))
        self.assertTrue(self.ticket.is_available())
        self.assertEqual(self.ticket.get_type_name(), "Ticket")
        self.assertIsNone(self.ticket.get_event())

    def test_setters(self):
        self.ticket.set_price(120.0)
        self.assertEqual(self.ticket.get_price(), 120.0)

        self.ticket.set_validity_period("2025-02-01", "2025-02-05")
        self.assertEqual(self.ticket.get_validity_period(), ("2025-02-01",
"2025-02-05"))

        self.ticket.update_availability(False)
        self.assertFalse(self.ticket.is_available())

    def test_display_details(self):
        detail = self.ticket.display_details()
```

```
        self.assertIn("Ticket ID: T001", detail)

class DummyTicket:
    def calculate_price(self, quantity):
        return 100 * quantity

class TestTicketTypes(unittest.TestCase):

    def test_single_race_pass(self):
        t = SingleRacePass("SinglePass", 50, "One race only", "2025-03-15",
"A1")
        self.assertEqual(t.calculate_price(2), 120)

    def test_weekend_package(self):
        t = WeekendPackage("Weekend", 150, "All races", "2025-03-10",
"2025-03-12", ["Qualifiers", "Main Race"])
        self.assertEqual(t.calculate_price(1), 190)

    def test_season_membership(self):
        t = SeasonMembership("Season", 300, "Full season", 2025, "Gold",
["Lounge", "Merch", "VIP Parking"])
        self.assertEqual(t.calculate_price(1), 345)

    def test_group_discount(self):
        base = DummyTicket()
        t = GroupDiscount("Group", 100, "Group deal", 5, 20, base)
        self.assertEqual(t.calculate_price(4), 400)  # No discount
        self.assertEqual(t.calculate_price(5), 400)  # Discount applied

class TestEvent(unittest.TestCase):

    def setUp(self):
        self.event = Event("E001", "Grand Prix", "2025-05-01", "Yas Marina",
100)

    def test_getters(self):
        self.assertEqual(self.event.get_event_id(), "E001")
        self.assertEqual(self.event.get_name(), "Grand Prix")
        self.assertEqual(self.event.get_date(), "2025-05-01")
        self.assertEqual(self.event.get_venue(), "Yas Marina")
        self.assertEqual(self.event.get_capacity(), 100)
        self.assertEqual(self.event.get_available_seats(), 100)

    def test_setters(self):
        self.event.set_name("New GP")
        self.assertEqual(self.event.get_name(), "New GP")

        self.event.set_date("2025-06-01")
        self.assertEqual(self.event.get_date(), "2025-06-01")
```

```
        self.event.set_venue("New Venue")
        self.assertEqual(self.event.get_venue(), "New Venue")

        self.event.set_capacity(80)
        self.assertEqual(self.event.get_capacity(), 80)

        with self.assertRaises(ValueError):
            self.event.set_capacity(-5)

        with self.assertRaises(ValueError):
            self.event.set_available_seats(200)

    def test_seat_management(self):
        self.assertEqual(self.event.check_availability(), 100)

        self.event.reduce_seats(10)
        self.assertEqual(self.event.check_availability(), 90)

        with self.assertRaises(ValueError):
            self.event.reduce_seats(100)

        self.event.increase_seats(5)
        self.assertEqual(self.event.check_availability(), 95)

class TestOrder(unittest.TestCase):
    def setUp(self):
        ticket1 = Ticket(1, 100.0, "2025-01-01", "2025-01-05")
        ticket2 = Ticket(2, 150.0, "2025-01-02", "2025-01-06")
        self.user = User(1, "testuser", "password123", "salt123", "Test User",
"test@example.com", "1234567890",
                         "123 Test St")

        self.order = Order(101, "2025-05-13", "pending", self.user)
        self.order.add_ticket(ticket1)
        self.order.add_ticket(ticket2)

class TestPayment(unittest.TestCase):

    def setUp(self):
        self.payment = Payment(payment_id=201, amount=250.0,
method="credit_card")

    def test_payment_fields(self):
        self.assertEqual(self.payment.get_payment_id(), 201)
        self.assertEqual(self.payment.get_amount(), 250.0)
        self.assertEqual(self.payment.get_method(), "credit_card")
        self.assertEqual(self.payment.get_status(), "pending")
        self.assertIsNone(self.payment.get_transaction_date())
```

```
    def test_status_setter(self):
        self.payment.set_status("processed")
        self.assertEqual(self.payment.get_status(), "processed")

    def test_process_payment(self):
        result = self.payment.process_payment()
        self.assertTrue(result)
        self.assertEqual(self.payment.get_status(), "processed")
        self.assertIsNotNone(self.payment.get_transaction_date())

    def test_verify_payment(self):
        self.payment.process_payment()
        self.assertTrue(self.payment.verify_payment())


class TestAdmin(unittest.TestCase):

    def setUp(self):
        self.admin = Admin(admin_id=1, username="admin", password="admin123",
access_level="full")

    def test_admin_fields(self):
        self.assertEqual(self.admin.get_admin_id(), 1)
        self.assertEqual(self.admin.get_username(), "admin")
        self.assertEqual(self.admin.get_password(), "admin123")
        self.assertEqual(self.admin.get_access_level(), "full")

    def test_login_success_and_fail(self):
        self.assertTrue(self.admin.login("admin", "admin123"))
        self.assertFalse(self.admin.login("admin", "wrongpass"))

    def test_access_level_change(self):
        self.admin.set_access_level("limited")
        self.assertEqual(self.admin.get_access_level(), "limited")


class TestSalesReport(unittest.TestCase):

    def setUp(self):
        self.report = SalesReport(report_id=1, start_date="2025-05-01",
end_date="2025-05-31")
        self.report.record_sale(date(2025, 5, 1))
        self.report.record_sale(date(2025, 5, 1))
        self.report.record_sale(date(2025, 5, 2))

    def test_sales_data(self):
        sales = self.report.get_sales_data()
        self.assertEqual(sales[date(2025, 5, 1)], 2)
```

```
        self.assertEqual(sales[date(2025, 5, 2)], 1)

    def test_generate_report(self):
        summary = self.report.generate_report()
        self.assertIn("2025-05-01: 2 ticket(s) sold", summary)
        self.assertIn("2025-05-02: 1 ticket(s) sold", summary)
        self.assertIn("Sales Report: 1 from 2025-05-01 to 2025-05-31", summary)


if __name__ == '__main__':
    unittest.main()
```

File Structure

Our ticket booking system uses Pickle to save data into two main files:

- users.pickle - Saves all user accounts and login information
- orders.pickle - Saves all ticket purchases and order details

Example of what inside the file

In the users.pickle:

```
[

    User with ID "U1715536789":

        - Username: "salma_almansoori"

        - Email: "salma@gmail.com"

        - Password: "(encrypted password)"

        - Full Name: "Salma Almansoori"

        - Phone: "055-1234567"

        - Address: "Abu Dhabi, UAE"

    ,

    User with ID "U1715536790":

        - Username: "hessa_alzaabi"

        - Email: "hessa@gmail.com"
```

```
        - Password: "(encrypted password)"
        - Full Name: "Hessa Alzaabi"
        - Phone: "050-9876543"
        - Address: "Dubai, UAE"]
```

In the orders.pickle:

```
[
   Order with ID "O1715536800":
        - Order Date: May 13, 2025
        - Customer: salma_almansoori
        - Status: "confirmed"
        - Tickets:
            * Ticket ID "T1715536801": $120.00 (Single Race Pass)
            * Ticket ID "T1715536802": $120.00 (Single Race Pass)
        - Total Amount: $240.00
   ,
   Order with ID "O1715536810":
        - Order Date: May 13, 2025
        - Customer: hessa_alzaabi
        - Status: "confirmed"
        - Tickets:
            * Ticket ID "T1715536811": $500.00 (Season Membership)
        - Total Amount: $500.00]
```

How We Save and Load Data

When someone registers, we:

- Create a new User
- Get all existing users from users.pickle
- Add the new user to the list
- Save the whole list back to users.pickle

When someone buys tickets, we:

- Create a new Order with tickets
- Get all existing orders from orders.pickle
- Add the new order to the list
- Save the whole list back to orders.pickle

Loading Data from Files:

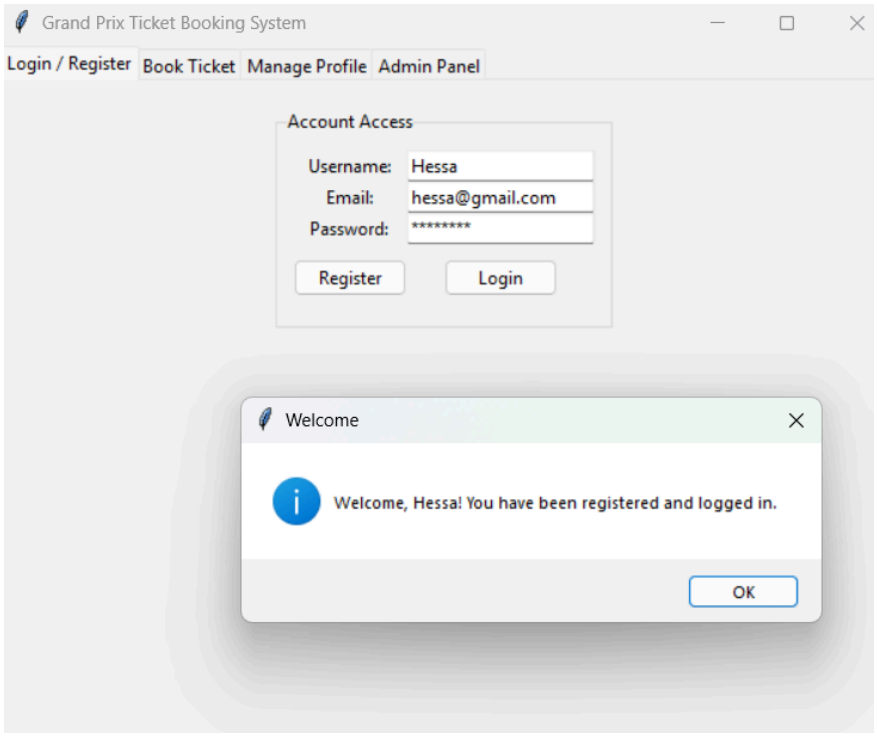
- Every time the app starts, it reads users.pickle to know who can log in
- When generating reports, it reads orders.pickle to show all sales
- If a file doesn't exist yet, we create an empty list

Code that handles file operations

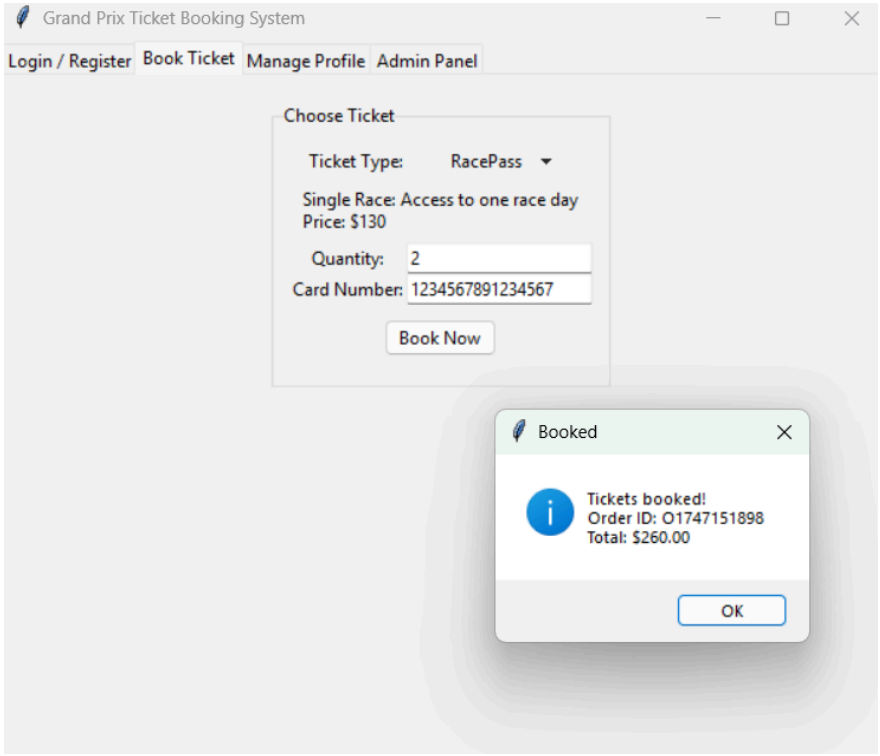
```
# This function reads data from pickle files

def load_data(filename):

    try:

        with open(filename, "rb") as f:

            return pickle.load(f)

    except (FileNotFoundError, EOFError):

        return []  # Return empty list if file is new


# This function saves data to pickle files

def save_data(filename, data):

    with open(filename, "wb") as f:

        pickle.dump(data, f)
```

GUI Screenshots

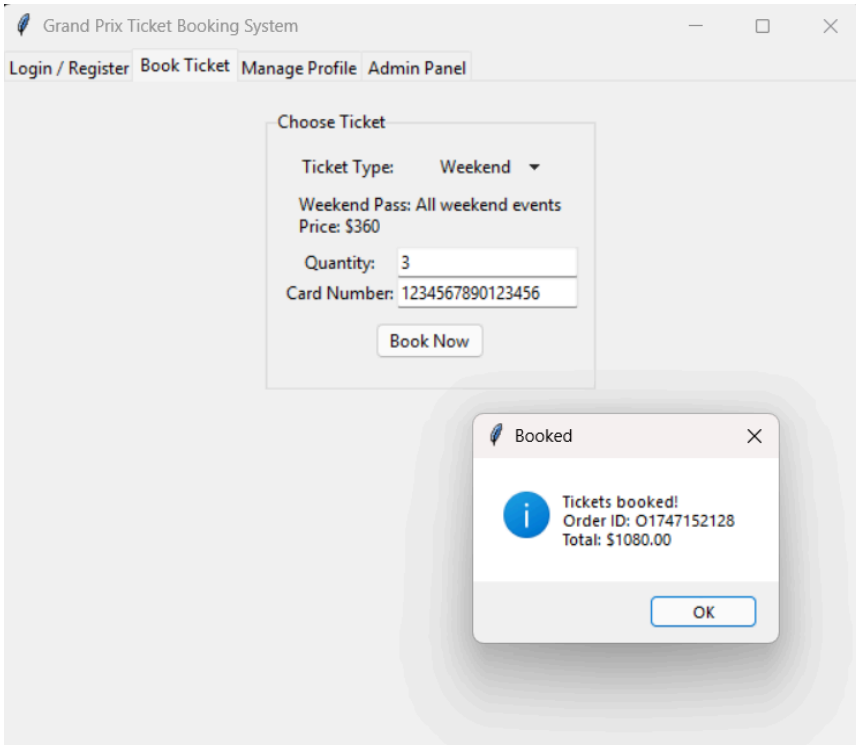
Registering and logging in



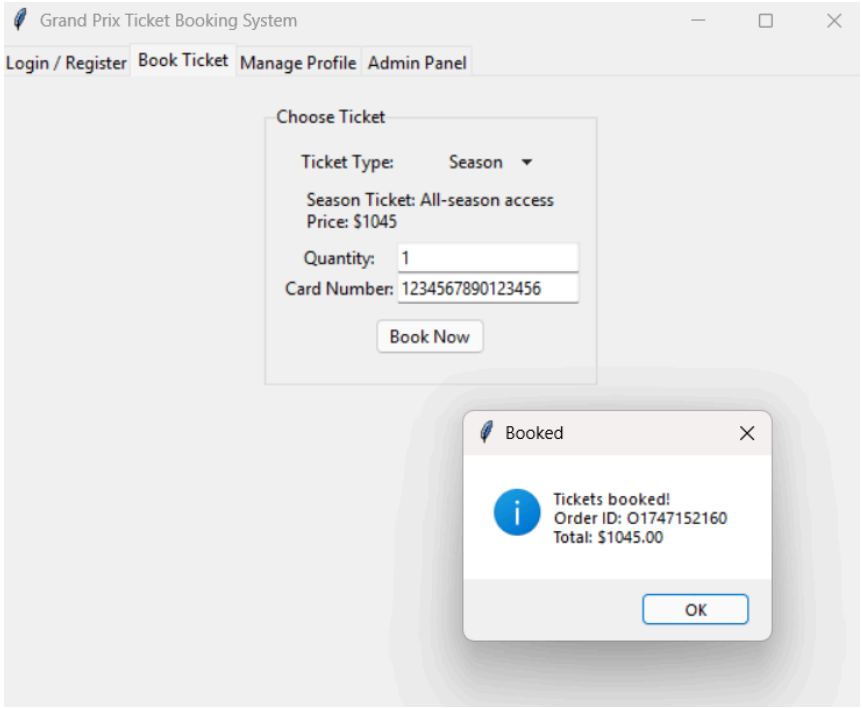
Booking 2 Single Race Car Tickets



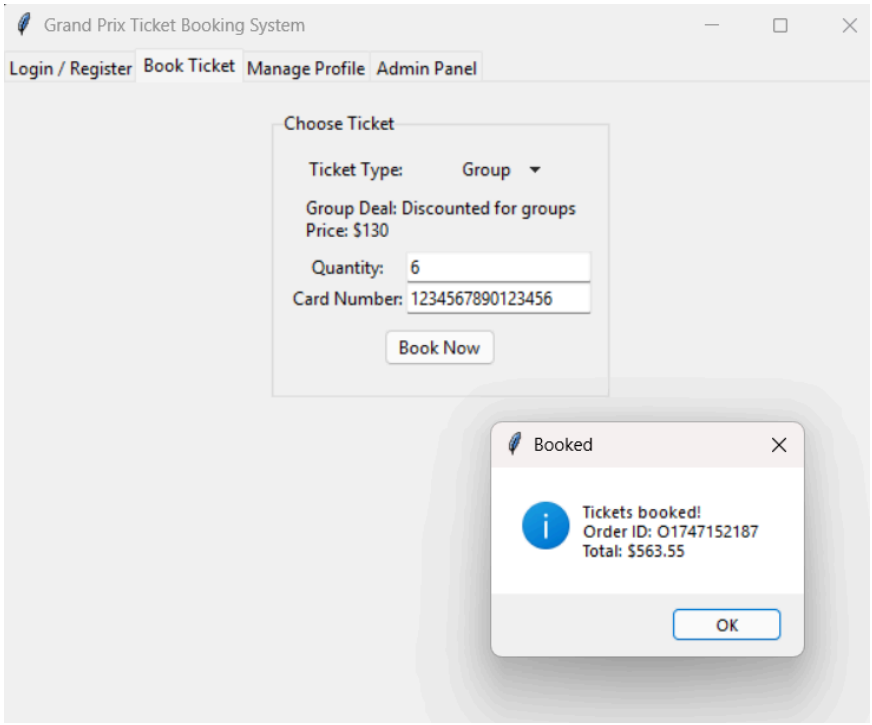
Booking 3 Weekend events tickets



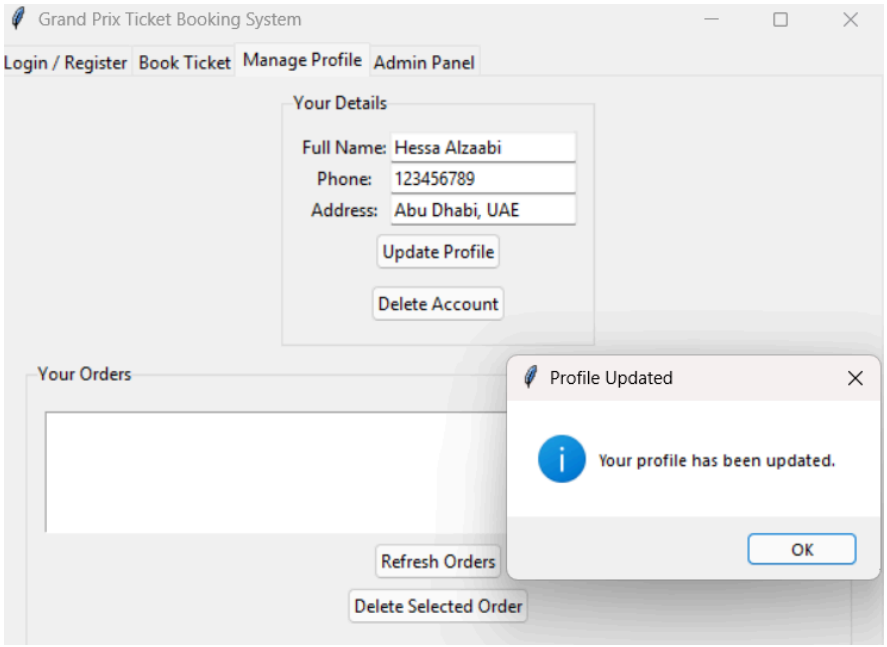
Booking 1 season ticket



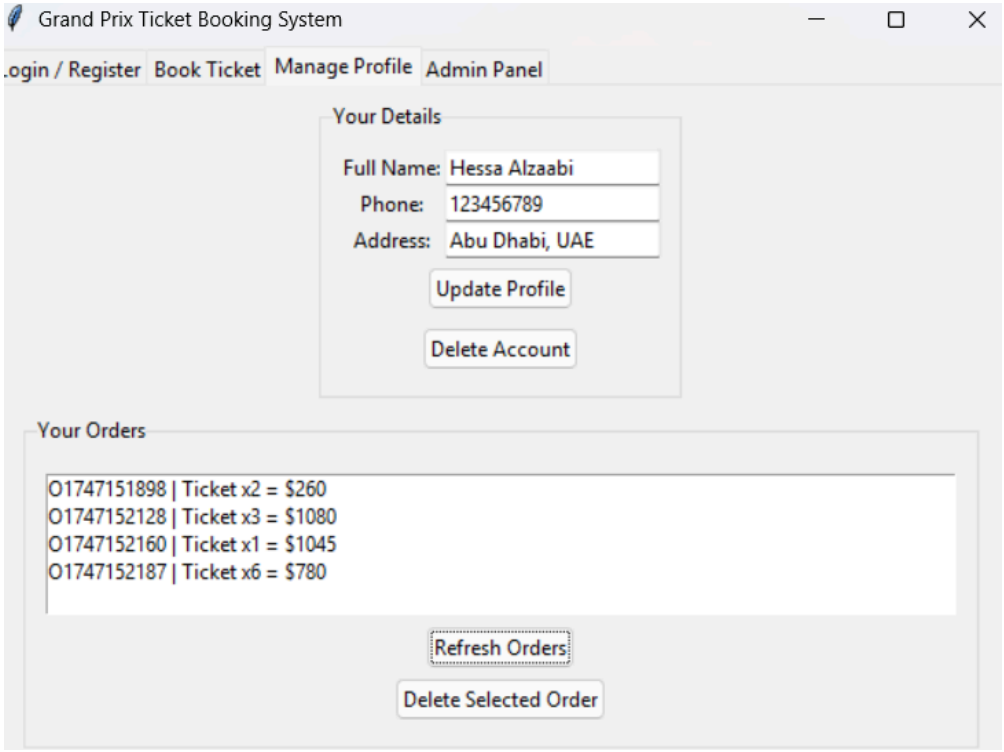
Booking 6 group tickets and getting a discount



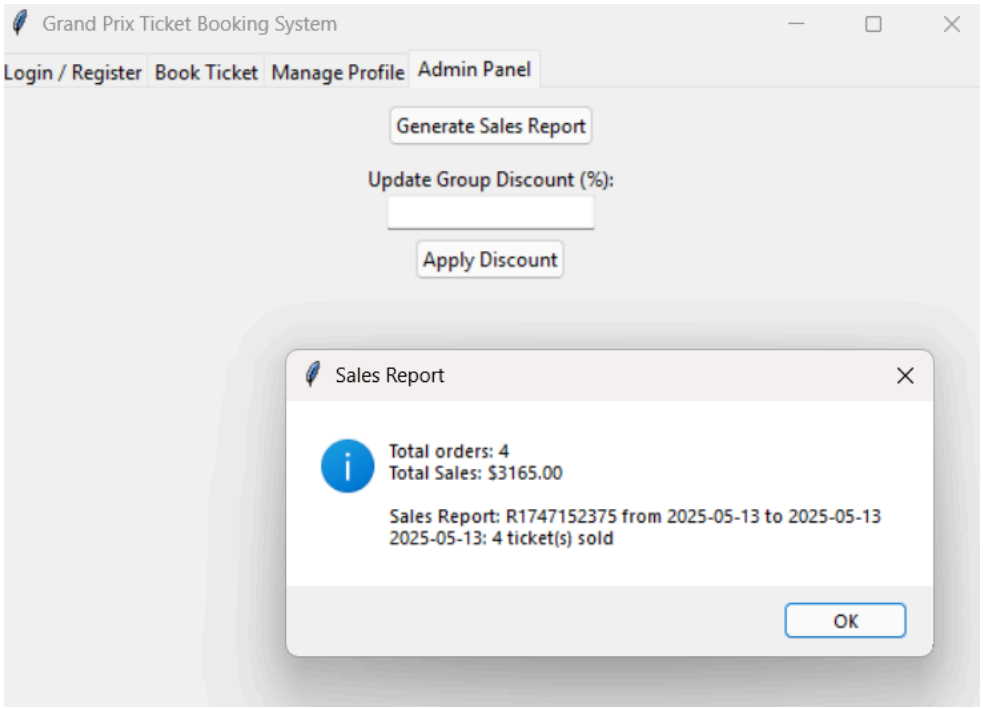
Updating the profile details in the Manage Profile tab



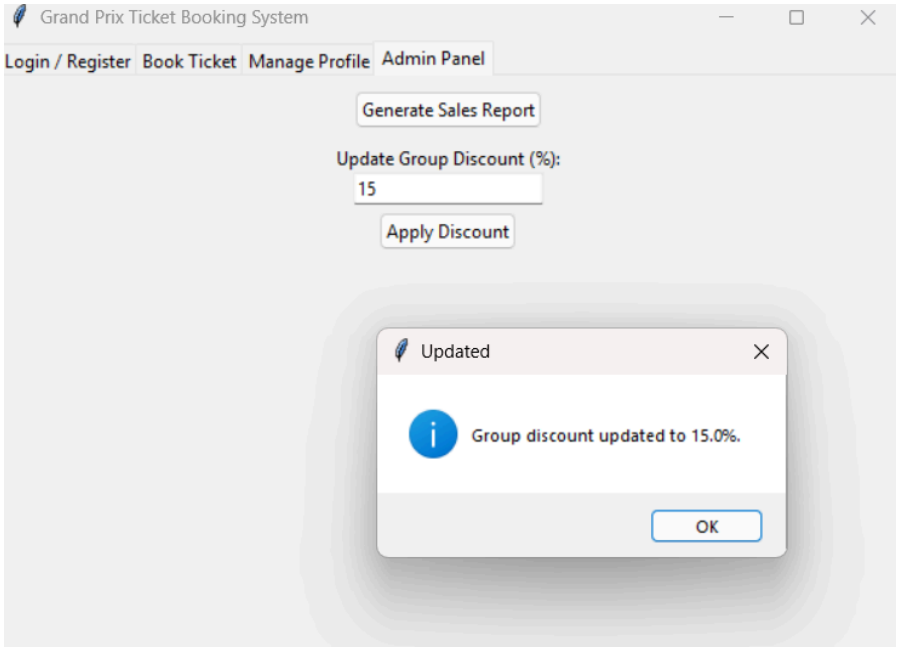
Refreshing the orders



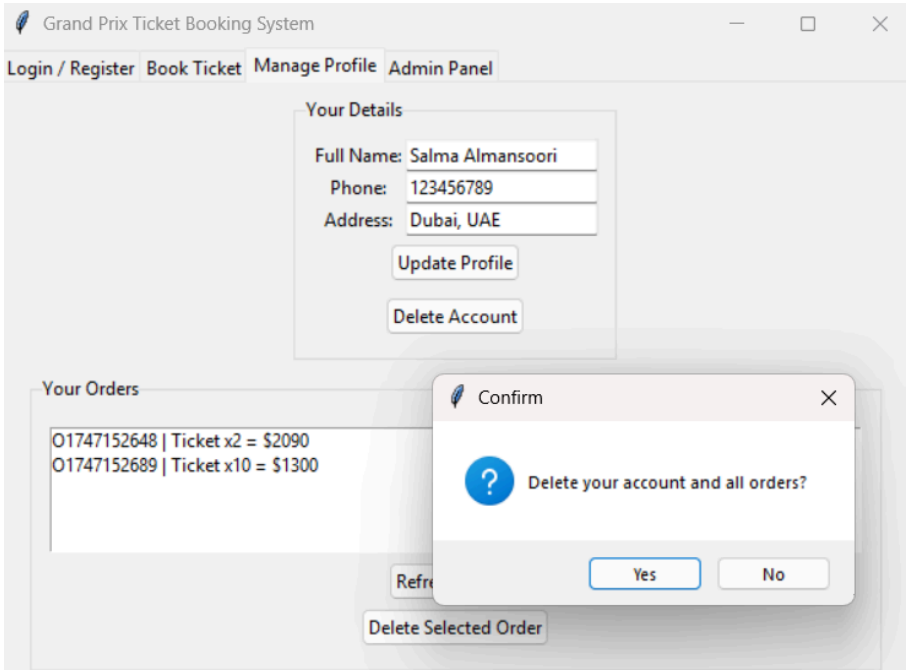
Generating the sales report in the Admin Panel tab

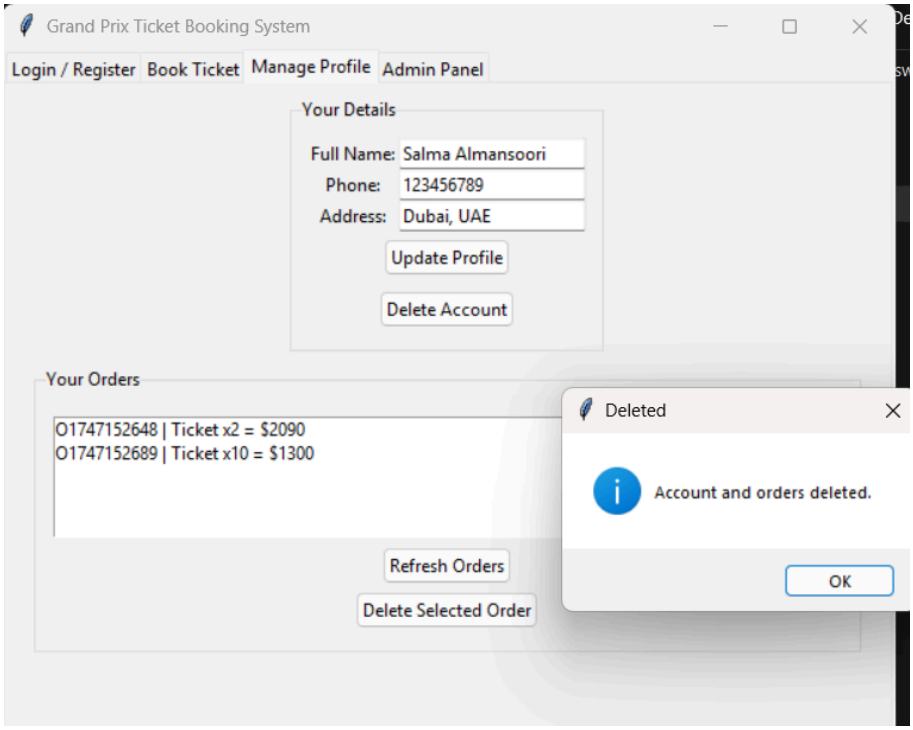


Updating the Group discount in the Admin Panel tab



Deleting an account and all of its orders





GitHub Repository Link

Github link: https://github.com/hessa-alzaabi/FinalProjectProgramming.git

Summary of Learnings

Salma Almansoori:

During this project, I learned a lot about designing software systems and turning ideas into real code. Creating the UML class diagram was like drawing a blueprint for our ticket booking system. I had to think carefully about what pieces we needed (like User, Ticket, Order classes) and how they connected to each other. Working with relationships between classes helped me understand how different parts of a system work together, like how an Order contains many Tickets and belongs to one User.

I also gained experience in Python programming, especially with object-oriented concepts like inheritance and encapsulation. Adding comments to the code made me realize how important it is to explain what each part does, so other people can understand it easily. Learning about Pickle was interesting, it's like saving your work in a special way that Python can read perfectly later.

The most rewarding part was seeing how our UML design became a working program with buttons and screens that people could actually use. This project taught me that good planning with UML diagrams makes coding much easier and helps create better software.

Hessa Alzaabi:

During the development of the Grand Prix Ticket Booking System, I was actively involved in multiple aspects of the project, which helped me strengthen both my technical and problem-solving skills. I wrote core modules and classes that handled user management, ticket types, event handling, payment processing, and administrative controls. This deepened my understanding of object-oriented design, modular programming, and system architecture. I worked on designing and capturing screenshots of the GUI using Tkinter. This helped me learn how to integrate backend logic with user interface components and enhanced my skills in layout design and event-driven programming. Also, I developed and fixed test cases for critical modules such as Payment, Admin, and SalesReport. This reinforced the importance of testing in software development and gave me hands-on experience in writing effective test cases using unittest. Overall, the project provided valuable experience in building a complete, testable Python application with a working GUI, data persistence, and user/admin functionalities.