Group Project Final Report

Project Title: Ride Booking System - Object Oriented Programming

Year / Section / Group Name / Number:

BSCPE 1-4 Group 7 - OOPS!

Team Members:

Name	Student ID	Role
Suzaine Mickaela Arquilla	2024-04564-MN-0	Technical Writer
Jermaine Jade De Guzman	2022-16985-MN-1	Full-Stack Developer
Mary Antoniette Ereve	2024-04508-MN-0	Technical Writer
Janelle Kate Fajardo	2024-05122-MN-0	Technical Writer
Juan Miguel Gamonido	2022-16986-MN-1	Full-Stack Developer
Trisha Faith Lleno	2024-02361-MN-0	Quality Assurance Tester
John Denver Villa	2024-03235-MN-0	UI/UX Designer

Instructor/Adviser:

Godofredo T. Avena

Date Submitted:

July 02, 2025

Table of Contents

8. Testing and Evaluation
9. Results and Discussion (Challenges)
10. Conclusion
11. References
12. Appendices

Introduction

The Ride Booking System was designed as a useful tool to simulate the essential features of ride-hailing services, providing consumers with an easy-to-use interface for booking and managing rides. Fundamentally, the project provides a means of putting object-oriented programming (OOP) concepts into practice, moving from abstract comprehension to practical application. With a graphical user interface (GUI) constructed with Tkinter and a Python development environment, it facilitates permanent data storage via file management.

The goal of developing this system was to create a working software that not only showcases technical mastery of Python but also replicates real-world systems where dynamic data manipulation, retrieval, and storage are required. Since ride-booking apps are a commonplace aspect of everyday life, this project gave the team the opportunity to recreate that experience in a more instructional, smaller-scale context.

It seeks to solve two issues: first, it provides users with an easy-to-use method of booking rides and viewing/managing their reservations without requiring a complicated setup or internet connection; second, it provides developers with a learning tool that demonstrates how OOP and GUI components can be combined to create a practical solution. The system tackles the problem of handling structured, permanent data in an interactive setting by mimicking a ride-booking experience, including vehicle selection, route management, cost computation, and booking cancellation. Thus, the Ride Booking System shows how fundamental programming concepts may be used to address real-world software development issues by fusing object-oriented design with responsive user interaction and data processing.

Objectives

GENERAL OBJECTIVE

To develop a Ride Booking System utilizing Object-Oriented Programming (OOP), incorporating file-handling for data persistence, and featuring a Graphical User Interface (GUI) built with Tkinter.

SPECIFIC OBJECTIVE

This Ride Booking System aims to;

1. Provide users with the ability to select their preferred vehicle type (motorcycle, car, or van) for booking.

2. Enable users to book, cancel, and view rides. This includes defining routes, confirming bookings and its details, and accessing a historical record of all ride activities.

Scope and Limitations

This Ride Booking System is a desktop application designed to simulate the core functionality of a ride-hailing service. Users can choose from pre-selected locations (which determine the distance automatically), manually enter a distance using the "Unlock Distance" option, or enter their name. There are at least three different vehicle types from which to choose, and each has its own pricing structure. Through a Treeview interface, the system allows users to create and cancel reservations as well as view all of their reservations. Additionally, it has a real-time filtering capability that facilitates user searches of booking records. Because all information is stored to a JSON file, reservations are kept safe even when the application is closed and then reopened.

The system does, however, have limitations. It does not support online or mobile platforms and is limited to desktop environments. All users are treated equally without personal accounts because there is no login or authentication feature. Vehicle availability is presumed to be limitless and always available; it is not tracked. It is not possible to modify the preset destinations from the GUI because they are hardcoded. Although users are able to manually enter distances, the system does not verify that these entries are accurate or realistic. Furthermore, no cloud features or other databases are used; storage is restricted to a local JSON file. Lastly, it is intended for use by a single user at a time on a local machine.

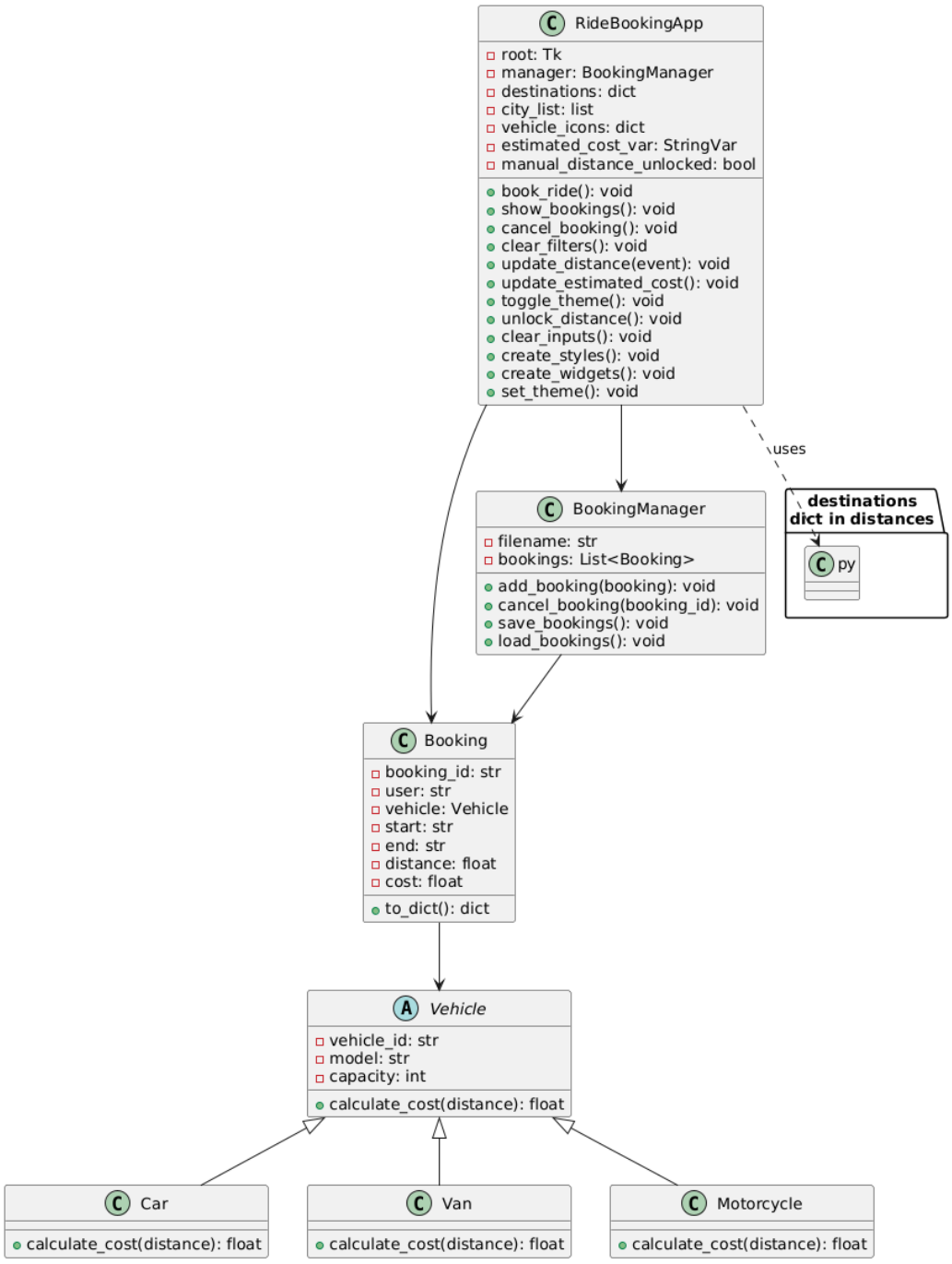
Methodology

The division and contribution of every member is divided into parts and sections. Jermaine Jade De Guzman and Juan Miguel Gamonido were in-charge of the coding of the functionality and framework of the whole program making sure that there's adequate development and proper flow once the program is used for implementation. The programmers made sure that every file will be able to interact with each other effectively and efficiently. John Denver V. Villa managed the designs and graphics of the program, making drafts and crafting ideas for the aesthetic of the program which makes it appealing and convincing to use. The designer makes sure that the program is marketable by making logos, button designs, and branding. He also made sure that everything is cohesive and made the designs very related towards each other. The paper and being the Quality Assurance Tester were assigned to Suzaine Mickaela Arquilla, Mary Antoniette Ereve, Janelle Fajardo, and Trisha Faith Lleno which makes them technical writers as they make sure that everything is flowing and systematic. They ensure that the program has no problems when it comes to implementation. They manage every document and provide detailed explanations. They

included every information and steps which makes the program more understandable and user friendly.

System Design

The system design shows the architecture and connections between the main elements of the ride booking system. The object-oriented structure is depicted in the class diagram below, which also shows how the main application communicates with managers and data, as well as connections with bookings and inheritance among vehicle kinds (car, van, and motorcycle).



Technologies Used

The Ride Booking System was developed using a collection of readily available and integrated technologies that facilitate user interaction, object-oriented design, and data durability. Because of its deliberate lightweight design and platform independence, the project can be easily executed on any desktop environment without requiring any further installations outside of Python.

- **Programming Language:** Python 3
- **GUI Library:** Tkinter
- **File Handling Format:** JSON
- **Programming Paradigm:** Object-Oriented Programming (Inheritance, Polymorphism, Encapsulation)
- **Platform:** Local Desktop Application (Windows/macOS/Linux)
- **Development Tools:** VS Code, IDLE, or any standard Python-compatible text editor
- **GUI Components:** Tkinter widgets, including ttk.Treeview for displaying booking data
- **Data File:** booking.js (automatically created and updated for persistent storage)

These technologies were chosen in order to blend simplicity and functionality. All things considered, the technology stack made it possible to develop a comprehensive and useful ride-booking simulation that captures important elements of real-world applications.

Implementation

This section covers the components, structure, and development process of the Python and Tkinter-based vehicle booking system. By choosing a vehicle type, entering personal information, and confirming the reservation, users can book rides through the system and choose their rider. All booking data is stored persistently in a JSON file/data format.

SYSTEM OVERVIEW

The booking system is a Python-based modular desktop application. Vehicle bookings are stored in a bookings.json file, and the Tkinter library is used to generate the interface. The system accommodates:

- Booking different types of vehicles (Car, Van, Motorcycle)
- Collecting customer data
- Retrieving and preserving bookings
- Using a management interface to display reservations
- The project is structured into separate modules to ensure maintainability and readability.

MODULE BREAKDOWN

main3.py

- Launches the GUI and handles user interaction.
- Uses Tkinter widgets to build the interface.
- Imports logic from vehicle, booking, and manager modules.
- Calls methods to create and store bookings.

manager.py

- Manages reading and writing to *bookings.json*.
- Contains functions for saving and retrieving booking data.
- Handles file operations and data formatting.

booking.py

- Defines the *Booking* class.
- Stores information such as customer name, vehicle type, contact, and pickup time.
- Converts booking objects to a dictionary for JSON storage.

vehicle.py

- Defines the vehicle classes used in the booking system:
  - **Vehicle**: A base class with shared attributes (*vehicle_id, model, capacity*) and an abstract *calculate_cost()* method.
  - **Car, Van, and Motorcycle**: Subclasses that override *calculate_cost()* based on their own pricing logic:
    - **Car**: distance * 10 + 50
    - **Van**: distance * 8 + 30
    - **Motorcycle**: distance * 5
- Represent the selected vehicle during booking and calculate the total cost dynamically based on distance.

Riders.py

- Handles random rider and vehicle selection for the booking system.
- Uses Tkinter to display names and vehicle plates on the GUI.
- Imports *random* module to randomly pick riders and plates.
- Displays labels with random data when the "Find" button is pressed.

distances.py

- Stores and manages distances between key destinations in Metro Manila.
  - **Destinations Dictionary:** A dictionary is defined with tuples representing pairs of cities and their corresponding distances (in kilometers).
  - **Purpose:** This module serves as a reference for the ride booking system to calculate the distance between two selected locations.

- **Key Feature:** It provides a straightforward way to access distances by querying the dictionary with city pairs, useful for route calculation or cost estimation based on distance.

bookings.json

- A persistent storage file used to save confirmed bookings in JSON format.
- Accessible by the manager module to display or modify bookings.

USER INTERFACE DESIGN

The GUI is designed using the Tkinter library, with additional modules like *ttk* for themed widgets.

- **Input Fields**: To input the user's name.
- **Dropdown Menus**: For selecting vehicle type, place of pickup, and destination.
- **Buttons**: To book a ride, view bookings, cancel selected booking, or toggle dark mode.
- **Images**: Displayed using *PIL.Image* to enhance user experience.

This interface is structured using Tkinter's *grid()* layout manager, providing a clean and organized design.

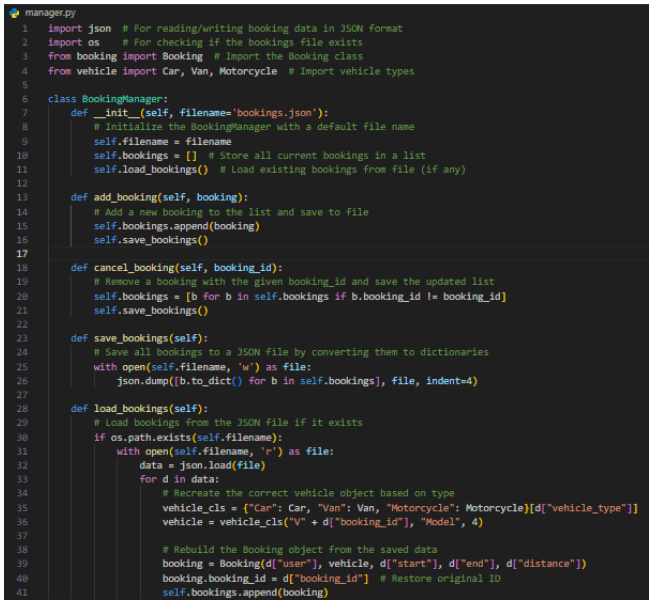
FEATURES DEVELOPED

Feature	Description
Vehicle Selection	Users can choose from Car, Van, or Motorcycle
Booking Form	Input fields and dropdown menus collect user data such as name, place of pickup, and destination
Booking Confirmation	Displays a message box containing users booking ID after a successful booking
Persistent Storage	Booking data is stored in *bookings.json*
View Bookings	Admins or users can view all saved bookings and admins' amount of earnings per booking.
View Receipt/Ticket	Users can view the receipt of their booking, including details like: ● Booking ID ● Vehicle type ● Total fare ● Driver's name and plate no.

	● Rebook and Exit Option

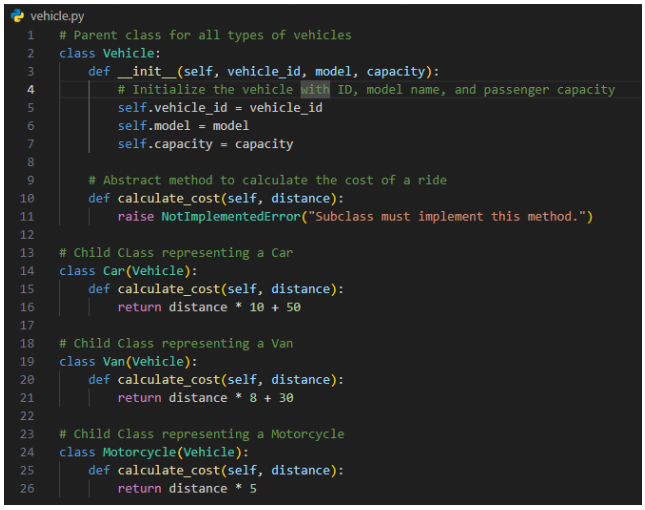
SAMPLE CODE SNIPPETS

manager.py



```python
import json  # For reading/writing booking data in JSON format
import os    # For checking if the bookings file exists
from booking import Booking  # Import the Booking class
from vehicle import Car, Van, Motorcycle  # Import vehicle types

class BookingManager:
    def __init__(self, filename='bookings.json'):
        # Initialize the BookingManager with a default file name
        self.filename = filename
        self.bookings = []  # Store all current bookings in a list
        self.load_bookings()  # Load existing bookings from file (if any)

    def add_booking(self, booking):
        # Add a new booking to the list and save to file
        self.bookings.append(booking)
        self.save_bookings()

    def cancel_booking(self, booking_id):
        # Remove a booking with the given booking_id and save the updated list
        self.bookings = [b for b in self.bookings if b.booking_id != booking_id]
        self.save_bookings()

    def save_bookings(self):
        # Save all bookings to a JSON file by converting them to dictionaries
        with open(self.filename, 'w') as file:
            json.dump([b.to_dict() for b in self.bookings], file, indent=4)

    def load_bookings(self):
        # Load bookings from the JSON file if it exists
        if os.path.exists(self.filename):
            with open(self.filename, 'r') as file:
                data = json.load(file)
                for d in data:
                    # Recreate the correct vehicle object based on type
                    vehicle_cls = {"Car": Car, "Van": Van, "Motorcycle": Motorcycle}[d["vehicle_type"]]
                    vehicle = vehicle_cls("V" + d["booking_id"], "Model", 4)

                    # Rebuild the Booking object from the saved data
                    booking = Booking(d["user"], vehicle, d["start"], d["end"], d["distance"])
                    booking.booking_id = d["booking_id"]  # Restore original ID
                    self.bookings.append(booking)
```

vehicle.py



```python
# Parent class for all types of vehicles
class Vehicle:
    def __init__(self, vehicle_id, model, capacity):
        # Initialize the vehicle with ID, model name, and passenger capacity
        self.vehicle_id = vehicle_id
        self.model = model
        self.capacity = capacity

    # Abstract method to calculate the cost of a ride
    def calculate_cost(self, distance):
        raise NotImplementedError("Subclass must implement this method.")

# Child Class representing a Car
class Car(Vehicle):
    def calculate_cost(self, distance):
        return distance * 10 + 50

# Child Class representing a Van
class Van(Vehicle):
    def calculate_cost(self, distance):
        return distance * 8 + 30

# Child Class representing a Motorcycle
class Motorcycle(Vehicle):
    def calculate_cost(self, distance):
        return distance * 5
```

Testing and Evaluation

Testing Methods:

1. **Unit Testing**
   Tests individual functions or components.
   Example: Checking if the booking logic correctly calculates the fare.

2. **Integration Testing**
   Tests combined parts of the app to ensure they work together.
   Example: Ensure that selecting a vehicle and confirming a booking updates the database or confirmation screen.

3. **System Testing**
   Tests the entire application for compliance with requirements.
   Example: Ensuring that all buttons, menus, and navigation flows function properly.

4. **GUI Testing**
   Specifically checks UI components in Tkinter.
   Example: Verifying if clicking the "Book Ride" button triggers the booking function.

5. **User Acceptance Testing (UAT)**
   Performed by intended users to verify that the app meets their needs.

Bug Tracking:

Exception Handling

In the **main3.py**, try-except blocks are used to handle potential errors, particularly those related to invalid user inputs. The first try-except block appears in the `update_estimated_cost` method, where it attempts to convert the input distance to a float. If the conversion fails, likely because the user entered a non-numeric value, then the program catches the exception and sets the estimated cost to "₱0.00" instead of crashing. Similarly, in the `book_ride` method, the code uses multiple try-except blocks. The first one again tries to convert the distance string to a float, and if it fails, a message box notifies the user that the distance is invalid. A second nested try-except block is used if the distance has not already been set and manual input is not unlocked; this block attempts to retrieve and store the distance between two destinations. If that fails due to an invalid input, it also notifies the user with an error message. Overall, these try-except structures ensure that the application remains stable and user-friendly even when incorrect or unexpected data is entered.

```
def update_estimated_cost(self):
    try:
        distance = float(self.distance_var.get())
    except:
        self.estimated_cost_var.set("₱0.00")
        return

    v_type = self.vehicle_var.get().split(" ", 1)[-1]
    rates = {"Car": 20, "Van": 30, "Bike": 10}
    rate = rates.get(v_type, 0)
    cost = distance * rate
    self.estimated_cost_var.set(f"₱{cost:.2f}")

def book_ride(self):
    user = self.name_entry.get()
    v_icon_label = self.vehicle_var.get()
    v_type = v_icon_label.split(" ", 1)[-1]
    start = self.start_var.get()
    end = self.end_var.get()
    distance_str = self.distance_entry.get()

    if not all([user, start, end, distance_str]):
        messagebox.showwarning("Missing Info", "Please fill in all fields.")
        return
```

```
    try:
        distance = float(distance_str)
    except:
        messagebox.showerror("Invalid Distance", "Please enter a valid distance.")
        return

    if not distance and self.manual_distance_unlocked:
        try:
            distance = float(self.distance_var.get())
            self.destinations[(start, end)] = distance
        except ValueError:
            messagebox.showerror("Error", "Invalid distance.")
            return

    vehicle_cls = {"Car": Car, "Van": Van, "Bike": Bike}[v_type]
    vehicle = vehicle_cls("ID", "Model", 4)
    booking = Booking(user, vehicle, start, end, distance)
    self.manager.add_booking(booking)
    messagebox.showinfo("Success", f"Booking ID: {booking.booking_id}")
    self.clear_inputs()
    self.show_bookings()
    self.estimated_cost_var.set("₱0.00")
```

Sample Test Cases and Results:

Unit Test Summary

Test ID	Module	Inputs	Expected Output	Status
UT-01	Vehicle.calculate_cost	Car, distance = 15.0	15 × 10 + 50 = 200	Passed
UT-02	Vehicle.calculate_cost	Van, distance = 14.5	14.5 × 8 + 30 = 146	Passed
UT-03	Vehicle.calculate_cost	Motorcycle, distance = 5.4	5.4 × 5 = 27	Passed
UT-04	BookingManager.add	One valid Booking object	Booking saved to list + JSON file	Passed
UT-05	BookingManager.cancel	Existing booking_id	Removed from list + JSON updated	Passed
UT-06	BookingManager.load	Pre-populated bookings.json	All bookings were rehydrated correctly	Passed
UT-07	update_distance logic	start="Pasig", end="Manila"	7.9	Passed
UT-08	update_distance logic	start="Unknown", end="Unknown" + Input Manually	Manual entry allowed	Passed

Manual UI Test Cases

TC ID	Scenario	Steps	Expected	Result
UI-01	Theme toggle	Click “Toggle Dark Mode” → click again	Theme colors switch in both directions	Passed
UI-02	Unlock the manual distance	Select two cities without preset distance → click “Input Manually”	Distance entry becomes editable	Passed
UI-03	Booking flow	Fill all fields → Book Ride	Success message with booking ID + entry cleared	Passed
UI-04	Filter bookings	After several bookings, filter by user ‘Juan”	Treeview shows only Juan’s bookings; summary updates	Passed
UI-05	Cancel via UI	Select a booking → click “Cancel Selected”	Booking disappears; summary decrements	Passed

Results and Discussion

OOPS! - Ride Booking System successfully met its core objectives, delivering a functional system built with Object-Oriented Programming (OOP) principles and a Tkinter-based Graphical User Interface (GUI).

The *Vehicle Management* features were fully implemented, fulfilling the requirement to provide users with a choice of vehicle types, a key feature for configuring their booking experience. Also, all vehicle details, including availability, were accurately stored and retrieved, demonstrating well-handling of data.

The *Booking Features* were also successfully integrated. Users can efficiently book, cancel, and view rides. The system confirms booking, updates vehicle availability, shows booking details, and provides a clear historical record of all transactions.

However, despite these successes, the development process encountered several challenges:

- Tkinter does not support gradient color that is supposed to be the color of the OOPS! system.
- Arrangement of elements such as the logo symbol of OOPS! system had difficulty in placing them on the GUI.

Conclusion

OOPS! Ride Booking System delivered a functional system prototype, providing invaluable experience in core software development. We gained practical expertise in Object-Oriented Programming (OOP) for structuring the system, designing the Graphical User Interface (GUI) using Tkinter for user interaction, and managing data persistence through file handling. With the challenges we encountered during development, it successfully deepened our understanding of real-world software engineering complexities and the crucial development for addressing real-world needs of many people.

Moreover, our effective teamwork and planning were crucial in overcoming development challenges. Through continuous collaboration, effective communication, and an adaptable approach, we were able to navigate and solve various technical challenges. Without our team's effort and good qualities, we wouldn't have managed to turn this project into a success.

Future Work

To further enhance the functionality, user experience, and real-world applicability of the OOPS! - Ride Booking System, the following features are recommended for future development:

1. Incorporate live map feature and expand the scope of location that can be booked (e.g., using Google Maps) to visually display the real-time location of available vehicles, the user's current position, and the progress of a booked ride.
2. Implement electronic payment options (e.g., GCash or PayMaya) and show the user's billings for more convenience in payment transaction experience.
3. Develop a user authentication system to allow for individual user accounts and drivers to configure and personalized their profile information in the system.
4. Integrate to mobile applications for both Android and iOS platforms. This would provide a more optimized and accessible user experience compared to a desktop-only GUI.

References

Code Nust. (2024, August 29). I CREATE RIDE BOOKING MANAGEMENT SYSTEM USING PYTHON & LEARN PYTHON BY BUILDING SIMPLE PROJECTS [Video]. YouTube. https://www.youtube.com/watch?v=3A_uQpoPKnA

Appendices

APPENDIX A

USER MANUAL

- **Opening the Application**
  1. Run the application

     welcome to OOPS!.py

  2. Click the "**BOOK NOW**" button to proceed with booking.
- **Making a Booking**
  1. Enter your **name**
  2. Select a **vehicle type** (Car, Van, Motorcycle)
  3. Choose a **pickup place** and **destination** from the dropdown.
  4. Click the **"Book Ride"** button
  5. Your booking ID (confirmation message) will appear if successful.
- **Choosing your driver**
  1. Select who your desired driver by clicking the "**Choose**" button.
  2. Click the "**FIND RIDERS**" button to choose/find other riders than what is shown on the screen.
- **Getting your receipt/ticket**
  1. A **message** "*Thank you for riding with us!*" and a **ticket/receipt** will appear on the screen after a successful booking.
  2. Click the "**Rebook**" button to book again and "**Exit**" button to close the application.
- **Viewing Bookings**
  1. Click "**View Bookings**" to see all confirmed bookings.
- **Exiting the Application**
  1. Click the *close* or *exit* button to close the application.

APPENDIX B

INSTALLATION GUIDE

- **Requirements**
  - Python 3.8+
  - Required libraries:
    - *tkinter*
    - *PIL*
- **Installation Steps**
  1. Download or clone the project folder.
  2. Go to the project directory and navigate.
  3. Install dependencies using pip on your command prompt:

     pip install Pillow

  4. Run the application

     welcome to OOPS!.py