Summative Assessment 2: System Implementation

Object Oriented Programming

Word count: 1534

Content

GitHub repository:

https://github.com/busilas/OOP_UoE/tree/main/Unit11/driverless-car-system

README

Word count: 666

This repository hosts a Python-based simulation of a driverless car system (DCS), leveraging Object-Oriented Programming principles. The system integrates crucial components such as environmental perception, passenger interaction, vehicle control, and navigation to demonstrate their collaboration in delivering a seamless and safe driving experience within a simulated environment.

Getting Started

To begin using the DCS, follow these steps:

1. **Clone the repository:**

```
git clone https://github.com/busilas/OOP_UoE/tree/main/Unit11/driverless-car-system
```

2. **Navigate to the Project Directory:**

```
cd driverless-car-system
```

3. **Run the application:**

```
python main.py
```

Interact with the system by following the prompts to log in, sign up, and navigate the car. Set destinations, initiate and halt journeys, and observe the vehicle's behavior.

Project Structure

The project is organized into several modules, each responsible for different aspects of the system:

1. **sensors.py** simulates various sensors used in the driverless car, including Lidar, Camera, and GPS.

   Lidar generates random distance data to represent a 360-degree scan.

   Attributes:

   - sensorType: set to "Lidar".

   Methods:

   - readSensorData(): returns a dictionary with 360 random distances..

   Camera simulates an RGB camera.

   Attributes:

   - cameraType: set to "RGB".

   Methods:

   - captureImage(): returns a dictionary with the camera type and a placeholder for the captured image..

   GPS simulates a GPS sensor.

   Attributes:

   - latitude: Latitude coordinate.
   - longitude: Longitude coordinate.

   Methods:

   - getCoordinates(): returns the current coordinates, incrementing them slightly each call to simulate movement.

2. **environment_perception.py** processes data from sensors and detects obstacles.

EnvironmentalPerception integrates Lidar, GPS, and Camera.

Attributes:

- lidarSensor, gps, camera, sensorData, obstacles.

Methods:

- processSensorData(): reads data from all sensors and stores it in sensorData.
- detectObstacles(): detects obstacles based on Lidar data.
- assessObstacleRisk(): assesses the risk of detected obstacles.
- getObstacles(): returns the list of detected obstacles.

3. **passenger_interaction.py** handles user interactions and passenger-related functionalities.

User represents a user with attributes for first name, last name, account name, and password.

Methods:

- verify_password(password): verifies the provided password.

PassengerInteraction manages destinations and journey status.

Attributes: destinations, journey_started.

Methods:

- set_destination(location, destination): sets a destination for a location.
- search_destination(location): searches for a destination by location.
- edit_destination(location, new_destination): edits a destination.
- delete_destination(location): deletes a destination.
- start_journey(), stop_journey(): managee journey status.

DriverlessCar - simulates a driverless car.

Attributes: carModel, carSpeed, steeringAngle.

Methods:

- set_car_speed(speed), get_car_speed(): manage car speed.
- set_steering_angle(angle), get_steering_angle(): manage steering angle.
- apply_brake(): stops the car.

4. **driverless_car_system.py** Integrates user management and passenger interaction into the main system.

   DriverlessCarSystem manages user accounts and their interactions with the system.

   Attributes: users, passenger_interaction, current_user.

   Methods:

   - login(), logout(), signup(): manage user sessions.
   - user_menu(), main_menu(): display menus for user interactions.

5. **navigation.py** handles route planning and navigation.

   Navigation manages route planning and waypoint navigation.

   Attributes: route, current_index.

   Methods:

   - plan_route(start, destination): plans a route from start to destination.
   - get_next_waypoint(): returns the next waypoint.
   - has_reached_destination(): checks if the destination has been reached.
   - calculate_distance(point1, point2): calculates the distance between two points.

6. **control.py** controls the vehicle based on sensor data and navigation.

   Control manages vehicle speed and steering to avoid obstacles and follow a route.

Attributes: vehicle, environmentalPerception, navigation, current_speed, steering_angle.

Methods:

- accelerate(acceleration), brake(deceleration), steer(steering_angle): manage vehicle dynamics.
- updateVehicleDynamics(): updates vehicle speed and steering angle.
- updateSensorData(): updates sensor data.
- detectAndAvoidObstacles(): detects and avoids obstacles.
- navigate(start, destination): plans a route and starts navigation.
- follow_route(): follows the planned route.

7. **main.py** - the entry point of the application.

   main() Initializes the DriverlessCar and Control instances and demonstrates the usage of navigation and control functionalities.

Future Enhancements

This project can be extended in several ways:

- Improved Sensor Simulations: Implement more realistic sensor data generation (Thakur & Mishra, 2024; Saoudi et al., 2023).
- Advanced Navigation Algorithms: Use real mapping data and more complex pathfinding algorithms (Silva et al., 2024; Szántó et al., 2023).
- User Interface: Develop a graphical user interface for easier interaction (Yan et al., 2023).
- Machine Learning Integration: Incorporate machine learning for better obstacle detection and avoidance (Manikandan et al., 2023; Ntakolia et al., 2023).

Conclusion

The DCS is a foundational framework for simulating and understanding the complexities involved in driverless car technologies. By integrating various modules for environmental perception, passenger interaction, vehicle control, and navigation, this project provides a comprehensive overview of how driverless cars operate and make decisions in real time.

Code Structure

DCS rely on a sophisticated interplay of hardware and software to operate without human intervention. The software is organized into modules, each responsible for different aspects of the vehicle's functionality. Key modules include sensors, environmental perception, passenger interaction, control systems, and navigation.

1. Sensors and Data Acquisition

The foundation of a DC’s operation lies in its ability to perceive the environment. This is achieved through an array of sensors, including Lidar, GPS, and cameras.

- **Lidar:** Lidar (Light Detection and Ranging) sensors emit laser beams to measure distances to surrounding objects. The data collected forms a detailed 3D map of the environment, enabling the vehicle to detect obstacles and navigate complex terrains.
- **GPS:** The GPS (Global Positioning System) provides real-time location data, helping the vehicle to determine its precise position on the globe. This information is crucial for navigation and path planning.
- **Cameras:** Cameras capture visual data from the car's surroundings, which is used for object recognition, lane detection, and traffic sign recognition.

The sensor module code is responsible for initializing these sensors, capturing data, and preprocessing it for further analysis. For example, a Lidar sensor class looks like this:

```python
import random

class Lidar:
    """
    Represents a Lidar sensor used for environmental perception.
    Attributes:
        sensorType (str): The type of sensor, which is Lidar.
    """
    def __init__(self):
        """
        Initializes a LidarSensor instance.
        """
        self.sensorType = "Lidar"

    def readSensorData(self):
        """
        Simulates reading Lidar sensor data.
        Returns:
            dict: A dictionary containing Lidar sensor data with sensor
            type and data.
        """
        lidarData = {
            "sensor_type": self.sensorType,
            "data": [random.uniform(0.0, 20.0) for _ in range(360)]
            # Simulate 360-degree Lidar scan
        }
        return lidarData
```

This code initializes a Lidar sensor and simulates reading data, which would then be processed to create a 3D map.

2. Environmental Perception

The environmental perception module processes the raw data from sensors to understand the surroundings. It involves obstacle detection, risk assessment, and environment mapping.

- **Obstacle Detection:** Using data from Lidar and cameras, the code identifies potential obstacles in the vehicle's path. This is critical for avoiding collisions.

- **Risk Assessment:** Once obstacles are detected, the software assesses the risk level and determines if avoidance maneuvers are necessary.
- **Environment Mapping:** Combining data from various sensors, the vehicle creates a comprehensive map of its environment, which is essential for navigation and decision-making.

Here's an example of how the environmental perception module process sensor data:

```
from sensors import Lidar, GPS, Camera

class EnvironmentalPerception:
    """
    Class representing environmental perception using sensors.
    Attributes:
        lidarSensor (Lidar): The Lidar sensor instance.
        gps (GPS): The GPS sensor instance.
        camera (Camera): The camera instance.
        sensorData (dict): Dictionary to store sensor data.
        obstacles (list): List to store detected obstacles.
    """
    def __init__(self):
        """
        Initializes an EnvironmentalPerception instance with sensor objects
        and data storage.
        """
        self.lidarSensor = Lidar()
        self.gps = GPS()
        self.camera = Camera()
        self.sensorData = {}
        self.obstacles = []

    def processSensorData(self):
        """
        Processes sensor data from Lidar, GPS, and Camera.
        Reads Lidar sensor data, obtains GPS coordinates, and captures an
        image using the camera.
        """
        lidarData = self.lidarSensor.readSensorData()
        gpsData = self.gps.getCoordinates()
        cameraImage = self.camera.captureImage()

        self.sensorData = {
            "lidar": lidarData,
            "gps": gpsData,
```

```
            "camera": cameraImage
        }

    def detectObstacles(self):
        """
        Detects obstacles based on Lidar sensor data.

        Filters Lidar distances to identify obstacles within a certain
        range.
        """
        if "lidar" in self.sensorData:
            lidarDistances = self.sensorData["lidar"]["data"]
            self.obstacles = [distance for distance in lidarDistances if
distance < 10.0]
        else:
            self.obstacles = []

    def assessObstacleRisk(self):
        """
        Assesses obstacle risk based on detected obstacles.

        Returns:
            str: A message indicating whether high risk is detected or no
            obstacles are present.
        """
        if self.obstacles:
            return "High risk detected. Avoidance maneuver required."
        else:
            return "No obstacles detected."
```

This code demonstrates how sensor data is processed and used to detect obstacles and assess risks.

3. Passenger Interaction

The passenger interaction module manages user-related functionalities such as user registration, login, and destination setting. This module ensures that passengers can interact seamlessly with the vehicle.

- **User Registration and Login**: Code is written to manage user accounts, allowing passengers to create profiles and log in securely.

- **Destination Management**: Passengers can set, edit, and delete destinations, which the vehicle will use to plan routes.

An example of user interaction code looks like this:

```
class User:
    """
    Represents a user in the driverless car system, with functionalities
    to manage user details, login, and registration.
    """
    next_id = 1  # Static variable for user ID

    def __init__(self):
        """
        Initializes a User instance, prompting the user to enter their
        details.
        """
        self.user_ID = User.next_id
        User.next_id += 1
        self.first_name = input('Enter your first name: ')
        self.last_name = input('Enter your last name: ')
        self.account_name = input('Enter an account name: ')
        self.password = input('Enter a password: ')
        self.email = input('Enter your email address: ')

    def __repr__(self):
        """
        Represent the User instance as a string.
        """
        return f'User ({self.user_ID}) {self.account_name}'

    def verify_password(self, password):
        """
        Verify if the provided password matches the user's password.
        Args:
            password (str): The password to verify.
        Returns:
            bool: True if the password matches, False otherwise.
        """
        return self.password == password
```

This code handles user registration, creating unique IDs and storing user details.

4. Control Systems

The control systems module is responsible for the actual movement of the vehicle, including speed control, steering, and braking.

- **Speed Control**: The software adjusts the vehicle's speed based on the environment and desired travel speed.
- **Steering**: The vehicle's steering angle is adjusted to navigate turns and avoid obstacles.
- **Braking**: The software controls the braking mechanism to slow down or stop the vehicle when necessary.

An example of a control system class looks like this:

```
from passenger_interaction import DriverlessCar
from environmental_perception import EnvironmentalPerception
from navigation import Navigation

class Control:
    """
    Controls a driverless car by adjusting speed and steering angle based
    on environmental perception and navigation.
    Attributes:
        vehicle (DriverlessCar): The driverless car instance to control.
        environmentalPerception (EnvironmentalPerception): Environmental
        perception module for obstacle detection.
        navigation (Navigation): Navigation module for route planning.
        current_speed (float): The current speed set for the vehicle in
        meters per second.
        steering_angle (float): The current steering angle set for the
        vehicle in degrees.
    """

    def __init__(self, vehicle):
        """
        Initializes a Control instance for a driverless car.
        Args:
            vehicle (DriverlessCar): The driverless car instance to
            control.
        """
```

```
        self.vehicle = vehicle
        self.environmentalPerception = EnvironmentalPerception()
        self.navigation = Navigation()
        self.current_speed = 0.0
        self.steering_angle = 0.0

    def accelerate(self, acceleration):
        """
        Accelerate the vehicle by adjusting the current speed.
        Args:
            acceleration (float): The acceleration value in meters per
            second squared.
        Raises:
            ValueError: If acceleration is a negative value.
        """
        if acceleration >= 0:
            self.current_speed += acceleration
        else:
            raise ValueError("Acceleration must be a positive value.")

    def brake(self, deceleration):
        """
        Apply brakes to slow down the vehicle by adjusting the current
        speed.
        Args:
            deceleration (float): The deceleration value in meters per
            second squared.
        Raises:
            ValueError: If deceleration is a negative value.
        """
        if deceleration >= 0:
            self.current_speed -= deceleration
            if self.current_speed < 0:
                self.current_speed = 0.0
        else:
            raise ValueError("Deceleration must be a positive value.")

    def steer(self, steering_angle):
        """
        Adjust the steering angle of the vehicle.
        Args:
            steering_angle (float): The new steering angle in degrees.
        Raises:
            ValueError: If the steering angle is outside the valid range [-
            30, 30] degrees.
        """
        if -30 <= steering_angle <= 30:
```

```
            self.steering_angle = steering_angle
        else:
            raise ValueError("Steering angle must be between -30 and 30
            degrees.")

    def updateVehicleDynamics(self):
        """
        Update the vehicle's speed and steering angle based on the current
        settings.
        """
        self.vehicle.set_car_speed(self.current_speed)
        self.vehicle.set_steering_angle(self.steering_angle)

    def updateSensorData(self):
        """
        Update sensor data by processing environmental perception.
        """
        self.environmentalPerception.processSensorData()

    def detectAndAvoidObstacles(self):
        """
        Detect and avoid obstacles based on environmental perception data.
        """
        self.environmentalPerception.detectObstacles()
        risk_level = self.environmentalPerception.assessObstacleRisk()

        if risk_level == "High risk detected. Avoidance maneuver
        required.":
            self.brake(5.0)  # Simulate braking to avoid collision
            self.steer(10.0)  # Simulate steering to avoid obstacles
```

This class handles the dynamic control of the vehicle, integrating with the environmental perception module to ensure safe navigation.

5. Navigation

The navigation module integrates GPS data with mapping software to plan and adjust routes dynamically. It ensures that the vehicle follows the optimal path to reach the set destination.

- **Route Planning**: Using GPS data, the software plans a route from the current location to the destination.

- **Dynamic Adjustment**: The vehicle continuously adjusts its route based on real-time traffic data and environmental conditions.

An example snippet for route planning might be:

```
class Navigation:
    """
    Manages navigation and path planning for the driverless car.
    Attributes:
        route (list): A list of waypoints representing the planned route.
    """

    def __init__(self):
        """
        Initializes a Navigation instance.
        """
        self.route = []

    def plan_route(self, start, destination):
        """
        Plan a route from the start location to the destination.
        Args:
            start (tuple): The starting coordinates (latitude, longitude).
            destination (tuple): The destination coordinates (latitude,
            longitude).
        Returns:
            list: A list of waypoints from start to destination.
        """
        # For simplicity, this function generates a direct path with dummy
        waypoints
        self.route = [start, destination]
        return self.route

    def get_next_waypoint(self):
        """
        Get the next waypoint in the planned route.
        Returns:
            tuple: The next waypoint coordinates (latitude, longitude), or
            None if route is empty.
        """
        if self.route:
            return self.route.pop(0)
        return None

    def has_reached_destination(self):
        """
```

```
        Check if the destination has been reached.
        Returns:
            bool: True if the route is empty, indicating destination
            reached; False otherwise.
        """
        return not self.route

    def calculate_distance(self, point1, point2):
        """
        Calculate the distance between two points.
        Args:
            point1 (tuple): The first point coordinates (latitude,
            longitude).
            point2 (tuple): The second point coordinates (latitude,
            longitude).
        Returns:
            float: The distance between the two points.
        """
        # Simplified distance calculation for demonstration purposes
        lat_diff = point2[0] - point1[0]
        lon_diff = point2[1] - point1[1]
        return (lat_diff**2 + lon_diff**2)**0.5
```

This code plans a route using GPS coordinates and allows for dynamic adjustments.

Challenges and Significance

The development of DC software presents numerous challenges, including ensuring safety, handling edge cases, and achieving real-time processing.

- **Safety**: The primary concern is passenger and pedestrian safety. The code must be rigorously tested to handle all possible scenarios.
- **Edge Cases**: The software needs to handle rare and unpredictable situations, such as sudden obstacles or extreme weather conditions.
- **Real-time Processing**: The vehicle must process vast amounts of data in real-time to make instantaneous decisions.

Despite these challenges, the significance of DC is immense. They promise to reduce traffic accidents, increase mobility for individuals unable to drive, and improve traffic flow efficiency.

Conclusion

The code behind DC modules is a marvel of modern engineering, combining advanced algorithms, real-time data processing, and robust safety measures. Each module, from sensors to control systems, plays a crucial role in the seamless operation of autonomous vehicles. As technology continues to advance, the sophistication and reliability of DC will only increase, heralding a new era in transportation. This intricate interplay of software and hardware showcases the remarkable potential of code to transform our world.

Unit test

Testing plays a crucial role in the software development process to ensure that the application works properly and meets the requirements. In unit testing, assertions are used to verify that the tested code behaves as expected. These statements validate the expected behavior by comparing the actual output with the expected value or condition. If an assertion fails, it indicates a potential issue in the code, highlighting the importance of assertions in ensuring the correctness and reliability of the software through automated testing. Unit tests presented in Table 1.

Table 1. DCS test results

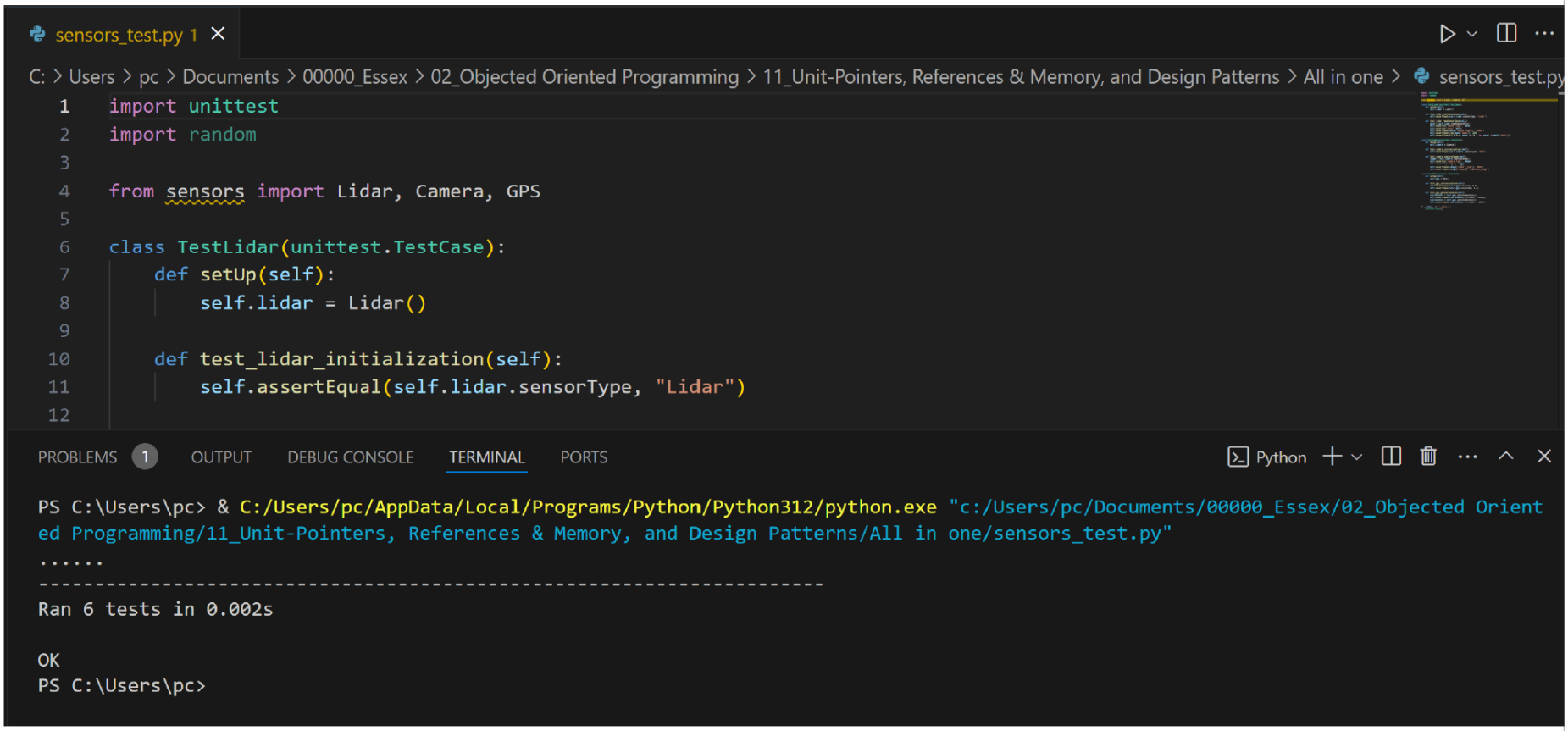
sensors_test.py

1. **TestLidar Class:**
    - **test_lidar_initialization:** Verifies the sensor type is correctly initialized.
    - **test_lidar_readSensorData:** Checks if the data returned by readSensorData method contains the correct keys, if the sensor_type is "Lidar", and ensures the data list has 360 float values within the range [0.0, 20.0].

2. **TestCamera Class:**
    - **test_camera_initialization:** Confirms the camera type is correctly initialized.
    - **test_camera_captureImage:** Ensures the captureImage method returns a dictionary with the correct keys and values.

3. **TestGPS Class:**
    - **test_gps_initialization:** Checks the initial values of latitude and longitude.
    - **test_gps_getCoordinates:** Tests the getCoordinates method to verify if it returns incremented latitude and longitude values correctly.

Console output:



passenger_interaction_test.py

1. **TestUser Class:**
    - **Uses** patch to mock input function calls for user details during initialization.
    - **test_user_initialization**: Verifies that the user attributes are correctly initialized.
    - **test_user_repr:** Checks the string representation of the User instance.
    - **test_verify_password:** Ensures the password verification works correctly.

2. **TestPassengerInteraction Class:**
    - **test_set_and_search_destination:** Tests setting and searching for destinations.
    - **test_edit_destination:** Verifies that destinations can be edited correctly.
    - **test_delete_destination:** Ensures destinations can be deleted.
    - **test_start_and_stop_journey:** Tests starting and stopping a journey.

3. **TestDriverlessCar Class:**
    - **test_initialization:** Confirms the car's model, speed, and steering angle are initialized correctly.
    - **test_set_and_get_car_speed:** Verifies setting and getting the car's speed.
    - **test_set_and_get_steering_angle:** Checks setting and getting the steering angle.
    - **test_apply_brake:** Tests that applying the brake sets the car's speed to zero.

Console output:

passenger_interaction_test.py 1

> Users > pc > Documents > 00000_Essex > 02_Objected Oriented Programming > 11_Unit-Pointers, References & Memory, and Design Patterns > All in one > passenger_interactio

```python
import unittest

from unittest.mock import patch
from passenger_interaction import User, PassengerInteraction, DriverlessCar

class TestUser(unittest.TestCase):
    @patch('builtins.input', side_effect=['John', 'Doe', 'johndoe', 'password123', 'john@example.com'])
    def setUp(self, mock_inputs):
        self.user = User()

    def test_user_initialization(self):
```

PROBLEMS 1 OUTPUT DEBUG CONSOLE TERMINAL PORTS Python

```
....Destination '123 Main St' set for location 'Home'.
Destination for location 'Home' deleted.
No destination found for location 'Work'.
.Destination '123 Main St' set for location 'Home'.
Destination for location 'Home' updated to '456 Elm St'.
No destination found for location 'Work'.
.Destination '123 Main St' set for location 'Home'.
.Journey started.
Journey is already in progress.
Journey stopped.
No journey is currently in progress.
...
----------------------------------------------------------------------
Ran 10 tests in 0.006s

OK
PS C:\Users\pc>
```

navigation_test.py

1. **TestNavigation Class:**
   setUp: Initializes a **Navigation** instance for each test case.

2. **test_initialization:**
   Verifies that the **route** attribute is initialized as an empty list.

3. **test_plan_route:**
   Tests the **plan_route** method by providing start and destination coordinates.
   Asserts that the route is correctly planned and assigned to the **route** attribute.

4. **test_get_next_waypoint:**
   Tests the **get_next_waypoint** method.
   Verifies that waypoints are correctly returned in sequence and returns **None** when the route is empty.

5. **test_has_reached_destination:**
   Tests the **has_reached_destination** method.
   Checks that it correctly identifies when the destination is reached based on the route being empty.

6. **test_calculate_distance:**
   Tests the **calculate_distance** method.
   Compares the calculated distance between two points with the expected value.

Console output:

navigation_test.py 1

Users > pc > Documents > 00000_Essex > 02_Objected Oriented Programming > 11_Unit-Pointers, References & Memory, and Design Patterns > All in one > navigation_test.py > ...

```python
import unittest

from navigation import Navigation

class TestNavigation(unittest.TestCase):
    def setUp(self):
        self.navigation = Navigation()

    def test_initialization(self):
        self.assertEqual(self.navigation.route, [])
```

PROBLEMS 1 OUTPUT DEBUG CONSOLE TERMINAL PORTS Python

```
PS C:\Users\pc> & C:/Users/pc/AppData/Local/Programs/Python/Python312/python.exe "c:/Users/pc/Documents/00000_Essex/02_Objected Oriented Programming/11_Unit-Pointers, References & Memory, and Design Patterns/All in one/navigation_test.py"
.....
----------------------------------------------------------------------
Ran 5 tests in 0.001s

OK
PS C:\Users\pc>
```

environment_perception_test.py

1. **TestEnvironmentalPerception Class:**
    - **setUp:** Initializes an **EnvironmentalPerception** instance for each test case. Mocks the sensor methods to return predefined values for consistent testing.
        - **Lidar:** Mocked to return 360 random float values between 0.0 and 20.0.
        - **GPS:** Mocked to return fixed coordinates (34.052235, -118.243683).
        - **Camera:** Mocked to return a fixed dictionary representing a captured image.

2. **test_initialization:**
    - Ensures that the sensor attributes are correctly initialized and that **sensorData** and **obstacles** are empty.

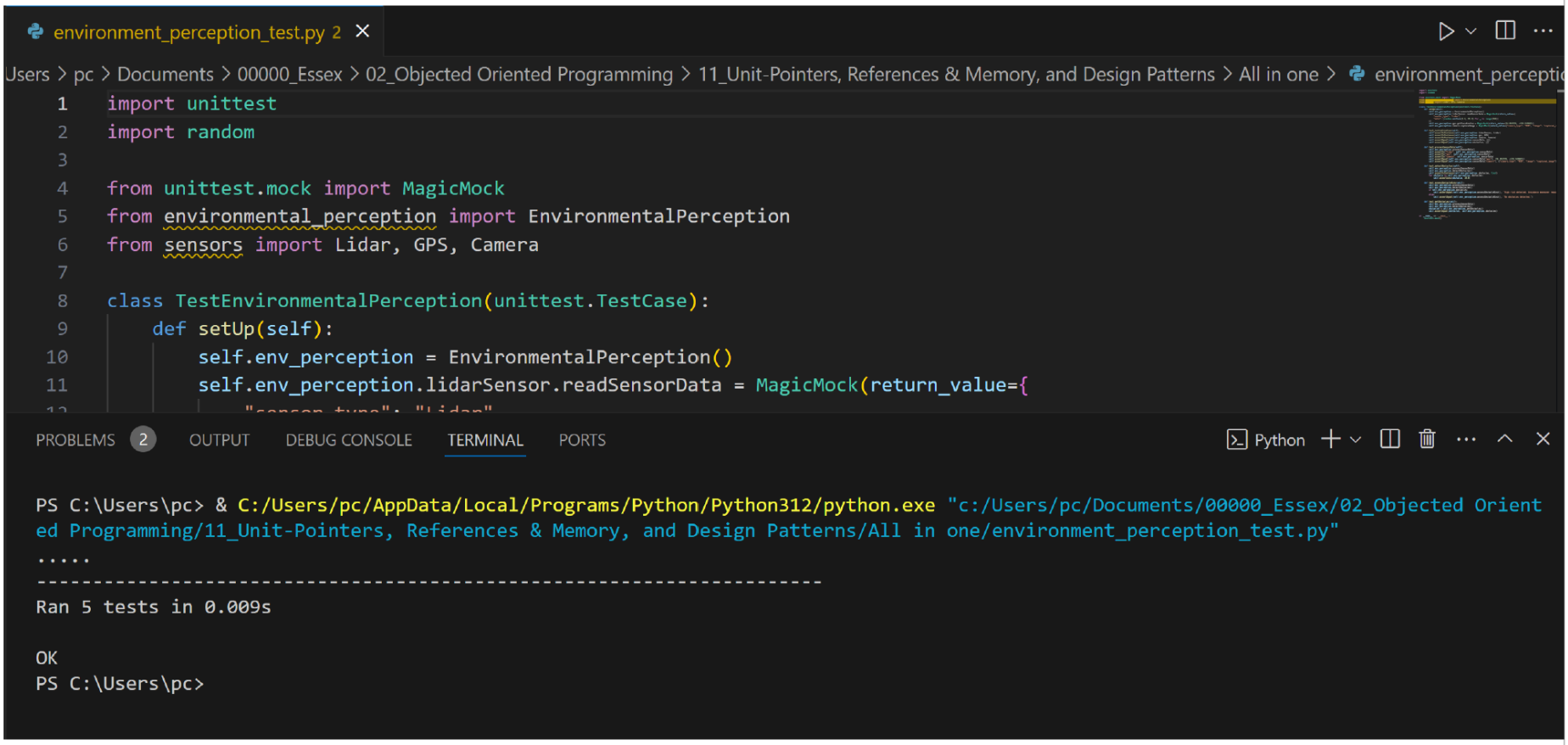
3. **test_processSensorData:**
    - Tests the **processSensorData** method to ensure it correctly populates **sensorData** with data from all sensors.

4. **test_detectObstacles:**
    - Tests the **detectObstacles** method to verify it correctly identifies obstacles within a 10.0 distance threshold based on the mocked Lidar data.

5. **test_assessObstacleRisk:**
    - Tests the **assessObstacleRisk** method to ensure it returns the correct risk message based on the presence of obstacles.

6. **test_getObstacles:**
    - Ensures that the **getObstacles** method correctly returns the list of detected obstacles.

Console output:



driverless_car_system_test.py_test.py

1. **TestDriverlessCarSystem Class:**
   - **setUp:** Initializes a **DriverlessCarSystem** instance for each test case.

2. **test_login_successful:**
   - Mocks user input for account name and password.
   - Adds a user with matching credentials.
   - Verifies successful login and appropriate message.

3. **test_login_user_not_found:**
   Mocks user input for a non-existent account name.
   Verifies that login fails with the correct message.

4. **test_login_incorrect_password:**
   - Mocks user input for account name and incorrect password.
   - Verifies that login fails with the correct message.

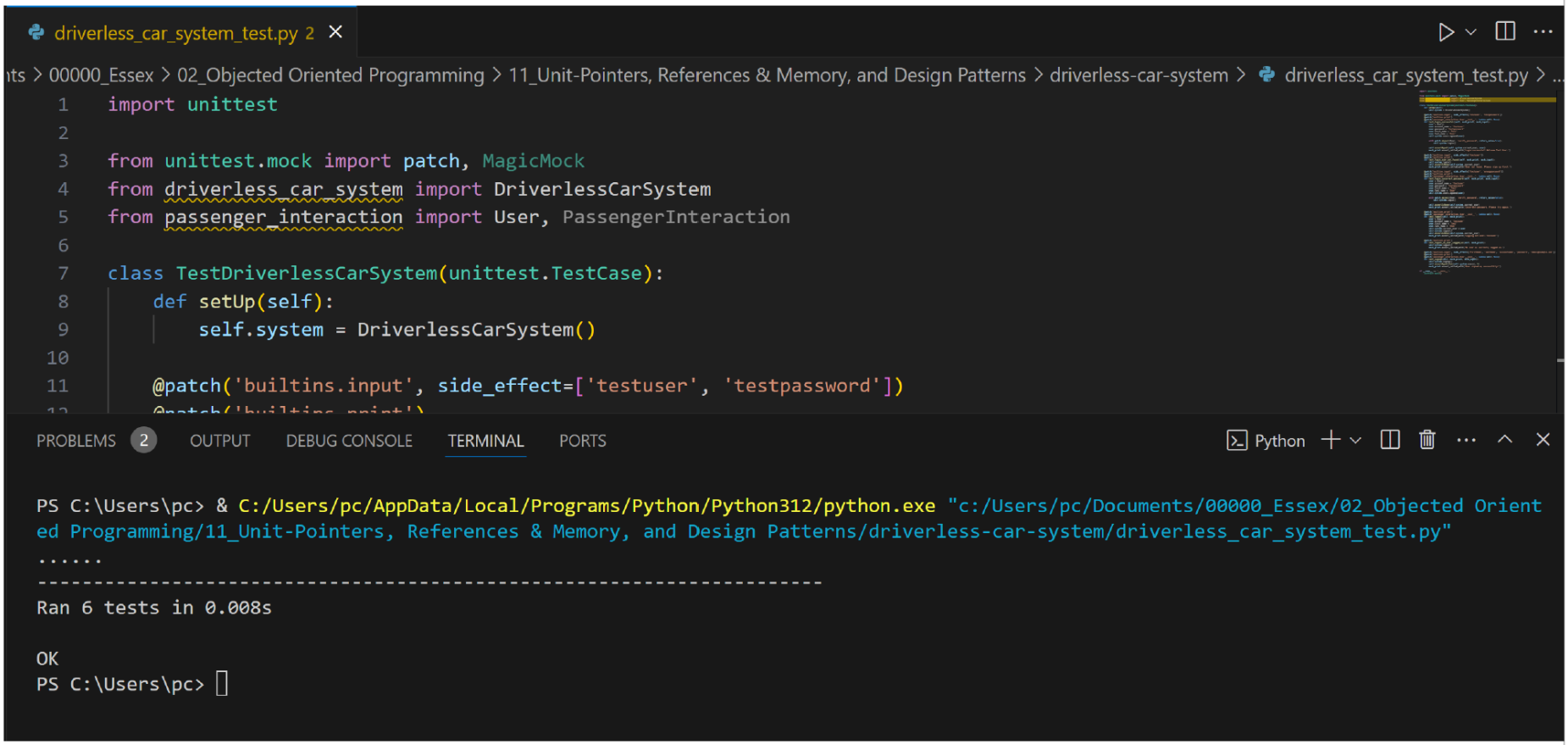
5. **test_logout:**
   - Sets a current user and tests logout functionality.
   - Verifies that the current user is set to None and the appropriate message is printed.

6. **test_logout_no_user_logged_in:**
   - Tests logout functionality when no user is logged in.
   - Verifies the correct message is printed.

7. **test_signup:**
   - Mocks user input for signup details.
   - Patches the **User.__init__** method to avoid actual input calls.
   - Verifies that a new user is added to the system and the correct message is printed.

8. **test_main_menu_quit:**
   - Mocks user input to quit the main menu.
   - Verifies that the system exits and prints the correct message.

Console output:



driverless_car_test.py

1. **TestDriverlessCar Class:**
    - **setUp:** Initializes a **DriverlessCar** instance with **ModelX** as the model and **50.0** as the speed before each test case.

2. **test_initialization:**
    - Verifies that the car model, speed, and steering angle are initialized correctly.

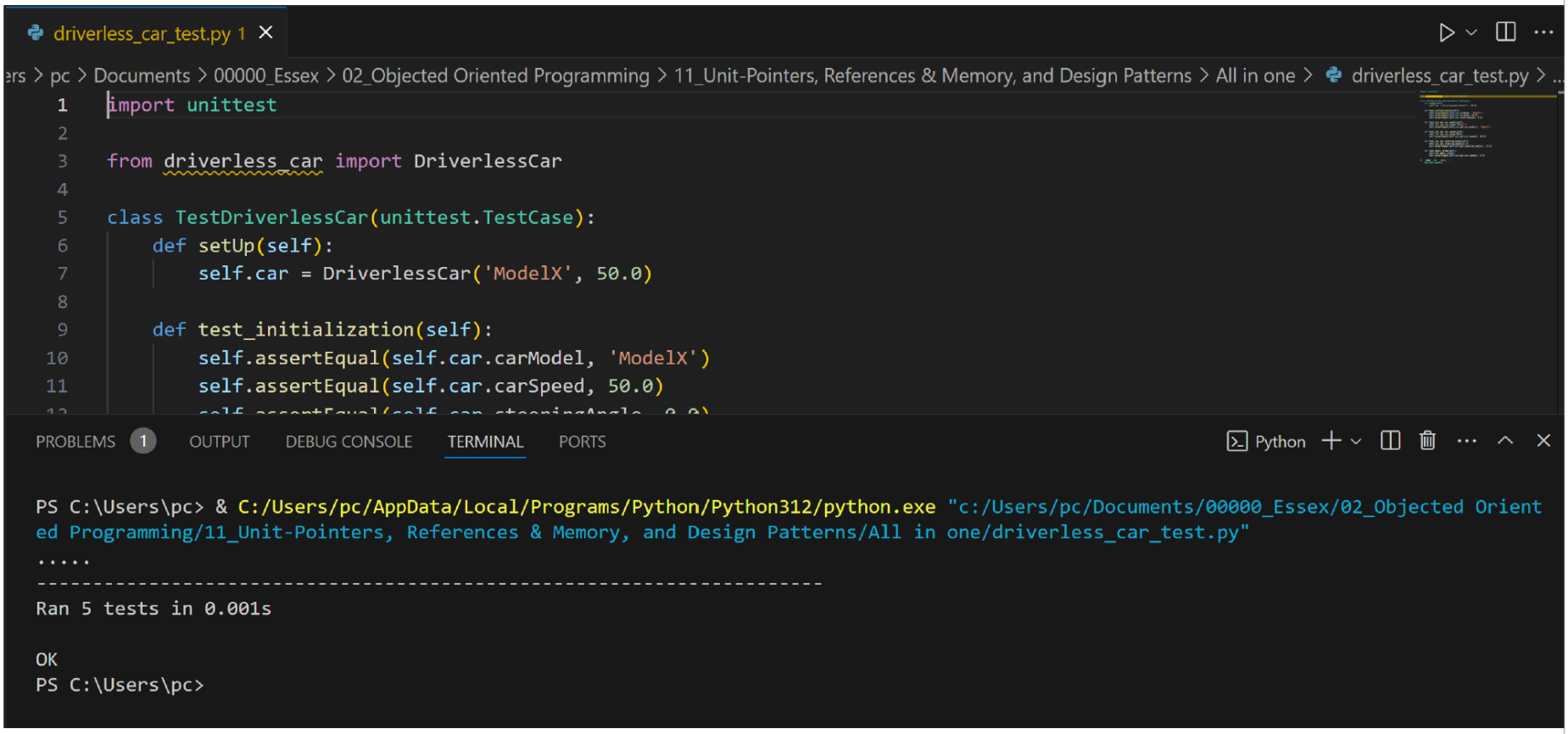
3. **test_set_get_car_model:**
    - Sets a new car model and verifies that the model is updated correctly using the getter method.

4. **test_set_get_car_speed:**
    - Sets a new car speed and verifies that the speed is updated correctly using the getter method.

5. **test_set_get_steering_angle:**
    - Sets a new steering angle and verifies that the angle is updated correctly using the getter method.

6. **test_apply_brake:**
    - Applies the brake and verifies that the car speed is set to **0.0**.

Console output:



control_test.py

1. TestControl Class:
    - setUp: Initializes a DriverlessCar instance and a Control instance before each test case.

2. test_accelerate:
    - Tests accelerating the vehicle and checks if the speed increases.
    - Checks for a ValueError when a negative acceleration value is provided.

3. test_brake:
    - Tests braking the vehicle and checks if the speed decreases appropriately.
    - Ensures speed does not go below zero and raises ValueError for negative deceleration.

4. test_steer:
    - Tests setting the steering angle within valid range.
    - Checks for a ValueError when an out-of-range steering angle is provided.

5. test_updateVehicleDynamics:
    - Uses patch to mock set_car_speed and set_steering_angle methods of DriverlessCar.
    - Verifies that the vehicle dynamics are updated correctly.

6. test_updateSensorData:
    - Uses patch to mock processSensorData method of EnvironmentalPerception.
    - Verifies that sensor data is updated.

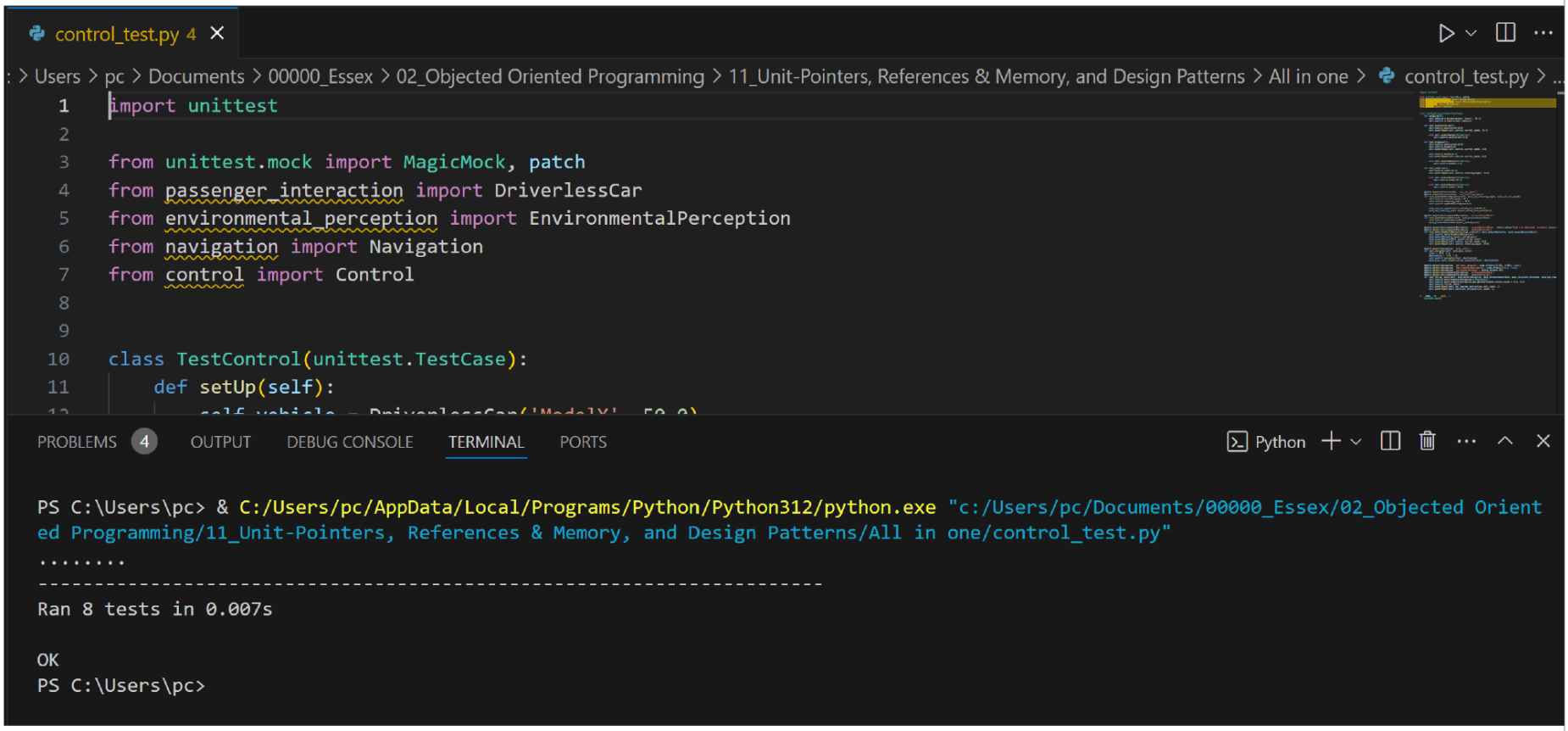
7. test_detectAndAvoidObstacles_high_risk:
    - Uses patch to mock detectObstacles and assessObstacleRisk methods of EnvironmentalPerception.
    - Verifies that the vehicle responds to high-risk obstacles by braking and steering.

8. test_navigate:
    - Uses patch to mock plan_route method of Navigation.
    - Verifies that the route is planned correctly.

9. test_follow_route:

- Uses multiple patch calls to mock methods and properties of Navigation and EnvironmentalPerception.
- Simulates following the planned route and verifies that the vehicle dynamics are updated appropriately.

Console output:



main_test.py

1. **Imports:**
    - **unittest:** The unit testing framework.
    - **patch, MagicMock:** For mocking objects and methods.
    - **DriverlessCarSystem:** The class we want to test.

2. **Test Class TestDriverlessCarSystem:**
    - **setUp:** Initializes an instance of DriverlessCarSystem before each test.
    - **test_main_menu_called:** Uses patch to mock the main_menu method and verifies it is called once when system.main_menu() is executed.
    - **test_driverless_car_system_initialization**: Asserts that the system instance is of type DriverlessCarSystem.

3. Main Block:
    - **unittest.main():** Runs the test suite when the script is executed directly.

Console output:

main_test.py 1

C: > Users > pc > Documents > 00000_Essex > 02_Objected Oriented Programming > 11_Unit-Pointers, References & Memory, and Design Patterns > All in one > main_test.py > ...

```python
import unittest

from unittest.mock import patch, MagicMock
from driverless_car_system import DriverlessCarSystem

class TestDriverlessCarSystem(unittest.TestCase):

    def setUp(self):
        self.system = DriverlessCarSystem()

    @patch.object(DriverlessCarSystem, 'main_menu')
```

PROBLEMS 1 OUTPUT DEBUG CONSOLE TERMINAL PORTS Python

```
PS C:\Users\pc> & C:/Users/pc/AppData/Local/Programs/Python/Python312/python.exe "c:/Users/pc/Documents/00000_Essex/02_Objected Oriented Programming/11_Unit-Pointers, References & Memory, and Design Patterns/All in one/main_test.py"
..
----------------------------------------------------------------------
Ran 2 tests in 0.003s

OK
PS C:\Users\pc>
```

Rationality of Changes for Summative Assessment 1: System Design

Table 2. DCS changes

No	Changes	Comment
1	Navigation and control separated as two modules according to the tutor feedback.	This provides a clearer representation of the system's functionality.
2	The passenger class was deleted and merged into the user class.	This helps to simplify and make the code easy to understand.

Reference list

Manikandan, N.S., Kaliyaperumal, G. & Wang, Y. (2023) ‘Ad hoc-obstacle avoidance-based navigation system using deep reinforcement learning for self-driving vehicles’, *IEEE Access*, 11, pp. 92285–92297. doi:10.1109/access.2023.3297661.

Ntakolia, C., Moustakidis, S. & Siouras, A. (2023) ‘Autonomous path planning with obstacle avoidance for Smart Assistive Systems’, *Expert Systems with Applications*, 213, p. 119049. doi:10.1016/j.eswa.2022.119049.

Saoudi, O., Singh, I. and Mahyar, H. (2023) ‘Autonomous vehicles: Open-source technologies, considerations, and development’, *Advances in Artificial Intelligence and Machine Learning*, 03(01), pp. 669–692. doi:10.54364/aaiml.2023.1145.

Silva, E., Soares, F., Souza, W. & Freitas, H. (2024) ‘A systematic mapping of autonomous vehicle prototypes: Trends and opportunities’, *IEEE Transactions on Intelligent Vehicles*, pp. 1–27. doi:10.1109/tiv.2024.3387394.

Szántó, M., Hidalgo, C., González, L., Rastelli, J., Asua, E. & Vajta, L. (2023) ‘Trajectory planning of automated vehicles using real-time map updates’, *IEEE Access*, 11, pp. 67468–67481. doi:10.1109/access.2023.3291350.

Thakur, A. & Mishra, S.K. (2024) ‘An in-depth evaluation of deep learning-enabled adaptive approaches for detecting obstacles using sensor-fused data in Autonomous Vehicles’, *Engineering Applications of Artificial Intelligence*, 133, p. 108550. doi:10.1016/j.engappai.2024.108550.

Yan, M., Rampino, L. & Caruso, G. (2023) ‘User acceptance of Autonomous Vehicles: Review and perspectives on the role of the human-machine interfaces’, *Computer-Aided Design and Applications*, pp. 987–1004. doi:10.14733/cadaps.2023.987-1004.

Summative Assessment 1: System Design

Summative Assessment 1: System Design

Object Oriented Programming

Word count: 963

A Design Proposal of Software to Support Operation of a Driverless Car

The concept of a driverless car (DC) refers to a vehicle, capable of navigating on an open road from point A to point B and requiring minimal input from a human driver or functioning entirely without one (Leiss, 2023; Bathla et al., 2022). DCs gather information from a variety of environmental recognition sensors and cameras such as laser, ultrasound, radar, and LIDAR. These passive systems as shown in Figure 1 allow for distance measurement and the processing of other information related to the vehicle's surroundings, as well as communication with infrastructure and other vehicles (Leiss, 2023). The collected data is processed by computer and used to control the engine acceleration, brakes, and steering, or to intervene in these systems as well as to locate paths, obstacles, and pertinent signage. This mimics the complex behaviour that human operators perform when they scan the path and car while driving. Some recent advancements in this field include the patent of traffic light interpretation by Google and Tesla's latest autonomous vehicle, which modifies its speed according to street signs. (Parekh et al., 2022).

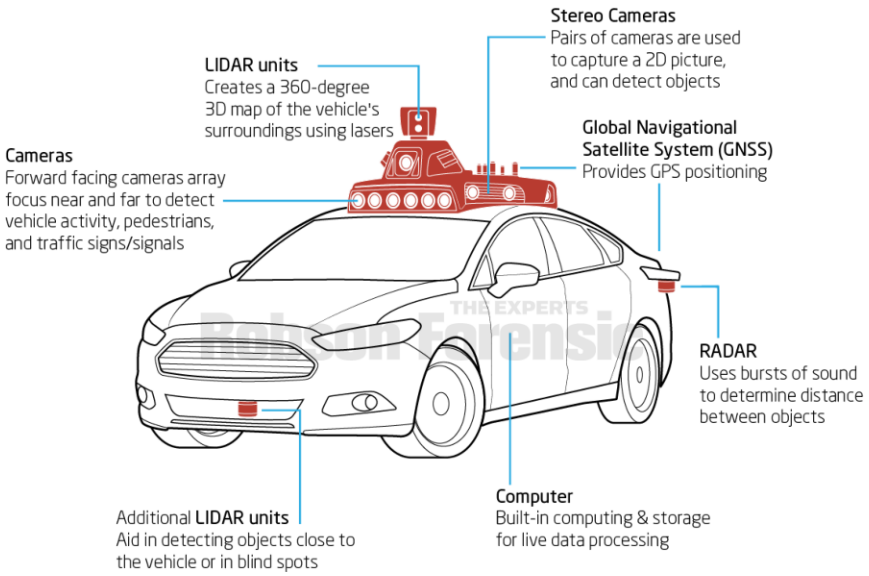


Figure 1. Basic components of driverless car (Leiss, 2023).

DCs may use various combinations of technologies to perform their autonomous functions. However, the functions outlined in Table 1 collaborate to enable a DC to operate securely and effectively in a range of driving circumstances. These functions, also known as the main functions of DC:

Table 1. Main functions of driverless car.

1. Environmental Perception	Environmental perception refers to developing a contextual understanding of the environment, such as where obstacles are located, detection of road signs/marking, and categorizing data by their semantic meaning (Pendleton et al., 2017). Primarily three types of sensors are in use today for data acquisition: 1. **LIDAR** – Light Detection and Ranging, is a remote sensing technology that leverages laser light to measure distances and construct maps of the environment. 2. **RADAR** - like LIDAR, is a technology that uses radio waves to detect the presence and location of objects 3. **Cameras** - capture images or video of the environment which can be used to detect and identify objects, such as pedestrians, other vehicles, traffic lights, and road signs.
2. Localization	The car determines its precise location and orientation on a map, often using GPS, inertial measurement units (IMUs), and other localization techniques. This module aims to localize the vehicle concerning its environment - know its position accurately enough to navigate to its target. It processes data from several of the sensors above and sometimes performs Simultaneous Localization and Mapping. GPS is often used to get a rough global estimate of the vehicle's position, and Kalman filters, particle filters etc. are used to fuse the information from the other sensors and get accurate localization (Thakurdesai & Aghav, 2020).
3. Path Planning	Based on the ***Environmental Perception*** and ***Localization*** data, the car plans a safe and efficient route to its destination, considering factors such as traffic conditions, speed limits, and road regulations. Path planning is a complex problem that deals with the physical constraints of DCs, other constraints from the environment, operational requirements, and above all, finding smooth paths (L'Afflitto et al., 2024).
4. Control and Navigation	The car's control system executes the planned path by controlling the steering, acceleration, and braking, while also navigating through traffic and following the planned route (Sjafrie, 2020).
5. Decision Making	The car's onboard computer system makes real-time decisions to handle complex driving scenarios, such as merging into traffic, changing lanes, and responding to unexpected events (L'Afflitto et al., 2024).
6. Communication	DCs may communicate with each other and with infrastructure (V2V and V2X communication) to share information about road conditions, traffic, and potential hazards (Bathla et al., 2022).
7. Passenger Interaction	Some DCs may provide interfaces for passengers to interact with the car's systems, monitor the route, and control certain functions (L'Afflitto et al., 2024).

UML Models

The UML diagram presented herein delineates the architecture and interactions of a driverless car system, encompassing crucial components responsible for autonomous vehicle operation and passenger interaction. This diagram specifically highlights the integration and collaboration of the four main modules: Environmental Perception, Localization, Control & Navigation, and Passenger Interaction. The purpose of the following UML diagrams is to offer a structured visualization of the manner in which these integral modules collaborate within the driverless car system, thereby demonstrating the flow of data and control necessary to ensure safe and efficient autonomous driving experiences.

1. Use case diagram

The use case diagram presented below depicts the three primary operations of Environmental Perception, Localization, Control and Navigation, as well as an additional operation for Passenger Interaction, all of which are included in the DC system.

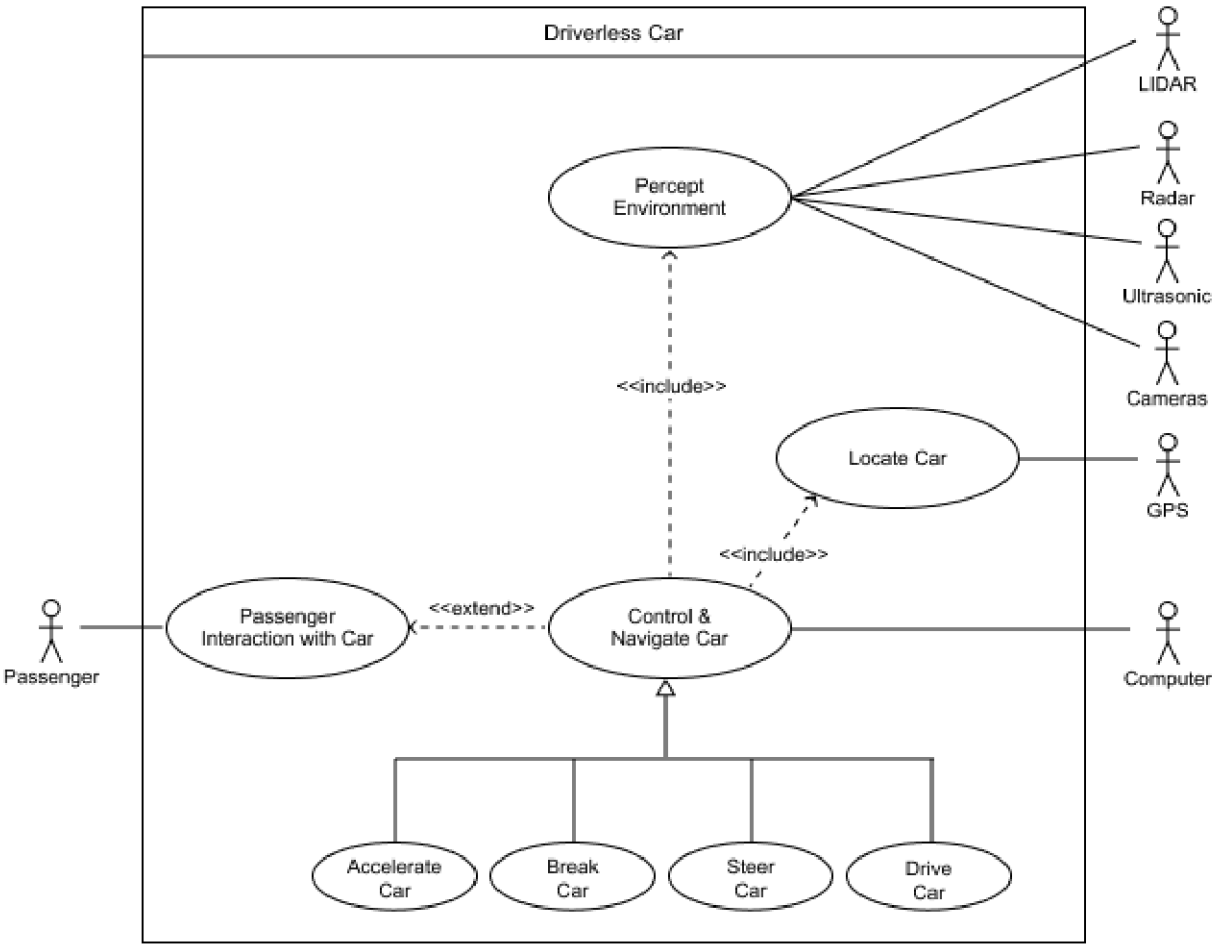


Figure 2. Use case diagram of DC.

2. Activity diagram

The activity diagram for the DC to drive to the destination, and navigate through any possible obstacles is illustrated in the figure shown below. It can be observed, that the following actions run parallelly: Generate coordinates values for the current environment, and Control and Generate local maps. Simple open-headed arrows depict the transition of control from one action to the next. Tiny squares with arrows and tiny squares joined by open-headed arrows (which are semantically equivalent) represent the movement of things, which can be data. Actions that include both incoming object and control flows are executed when both the control and the object are present.

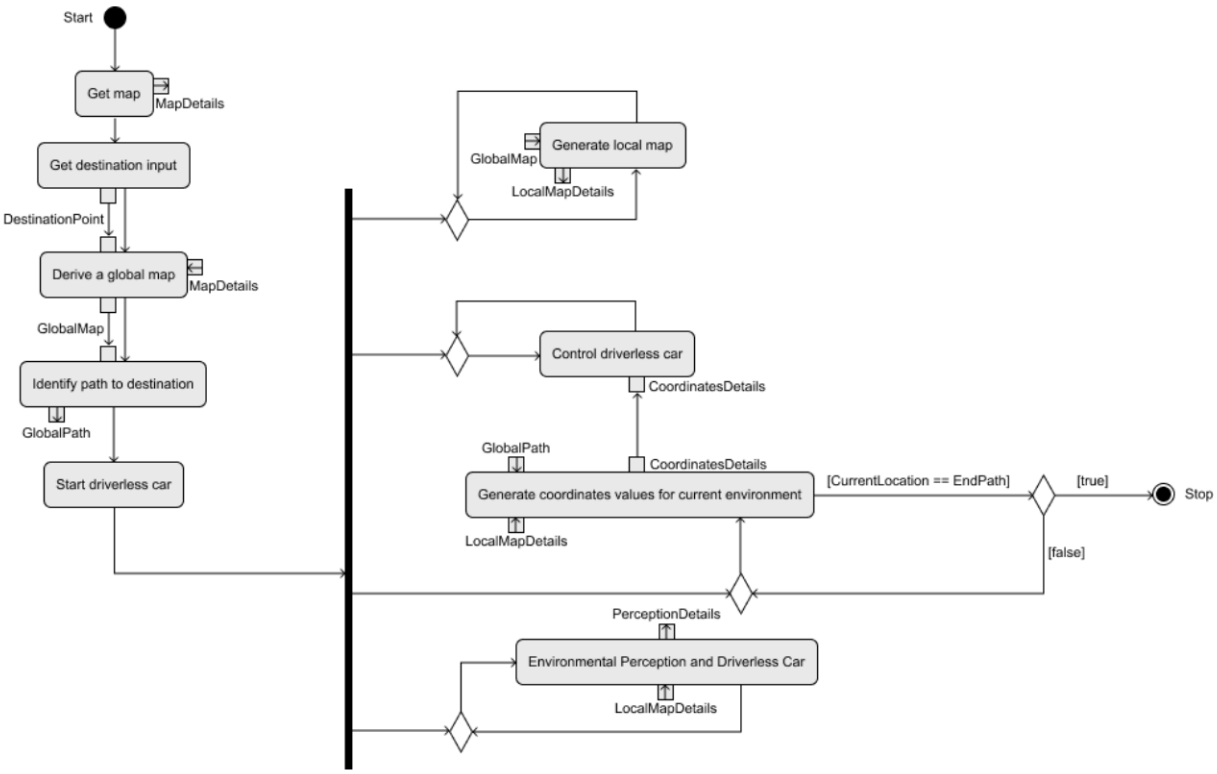


Figure 3. Activity diagram of full DC drive.

3. Class diagram

Figure 4 shows a UML class diagram that previews DC software with the relevant processes, including the required classes, subclasses, and relationships for implementation chosen operations (Environmental Perception, Localization, Control and Navigation, and additional operation of Passenger Interaction).

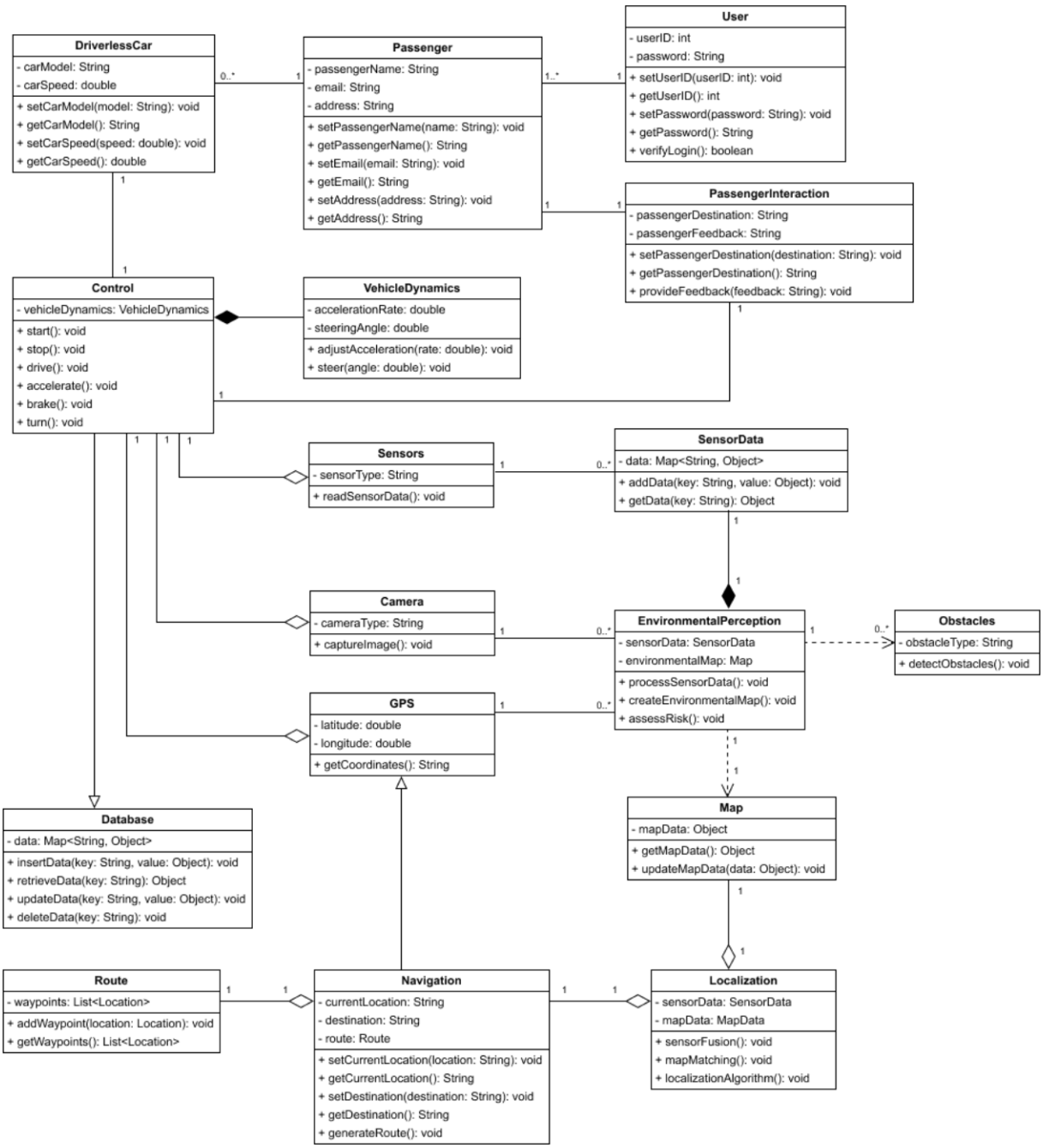


Figure 4. Class diagram

4. Sequence diagram

The sequence of interactions by setting the destination between the passenger and interactions of the DC system with other subsystems are shown in the sequence diagram in Figure 5. And Figure 6 represents sequence diagram of adjusting direction interacting between Controller, Database, VehicleDynamic and Sensosrs.

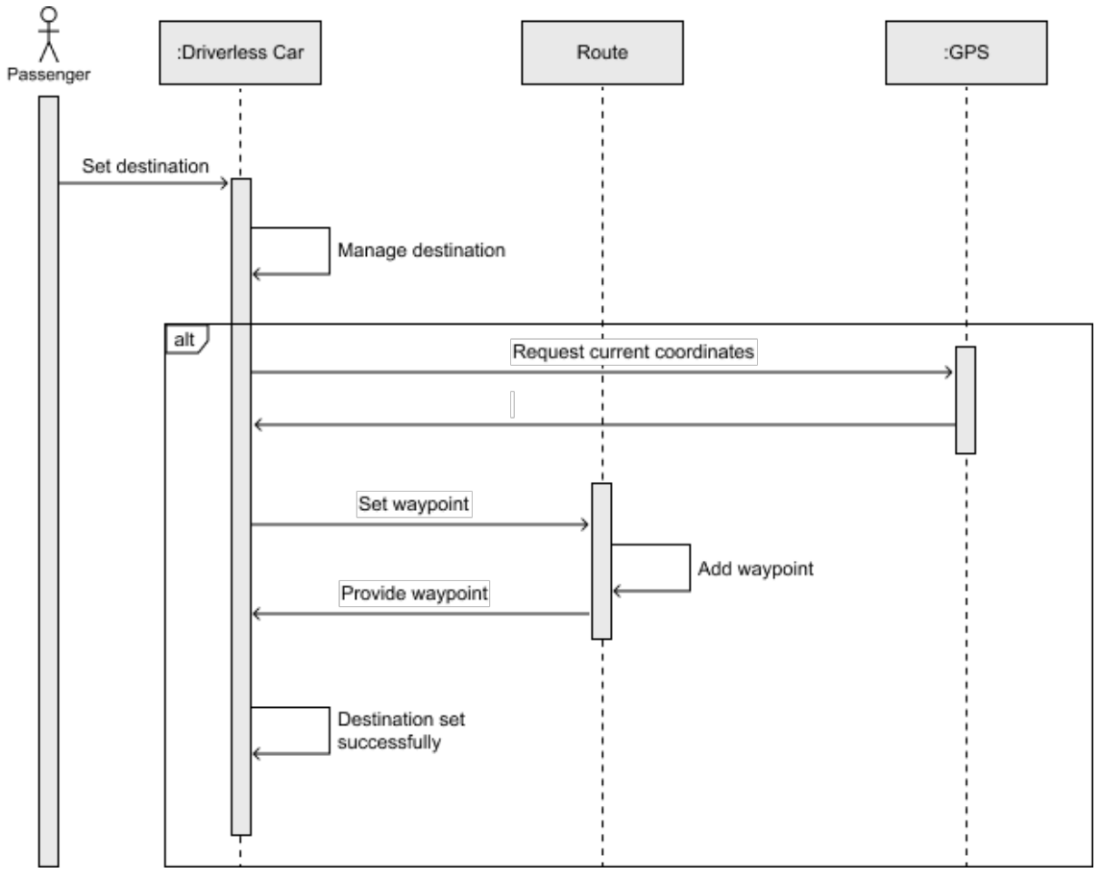


Figure 5. Set destination sequence diagram

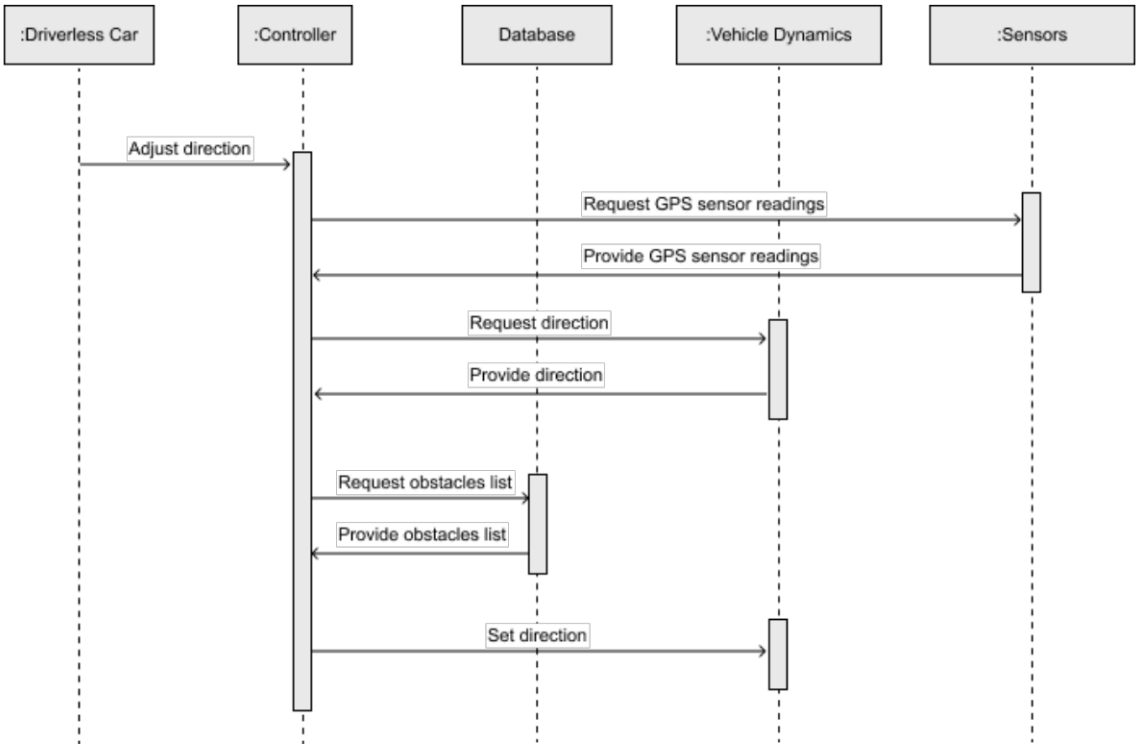


Figure 6. Adjust direction sequence diagram

5. State Transition Diagram

Figure 7 illustrates the route updates that occur when the path predicted by the algorithm differs from the actual route. If no data structure holds for the routes, every route must be checked to determine whether it is still possible according to the user's last action. These structures allow for the removal of previously established routes when several possible paths are synchronized with the past movement of the user.

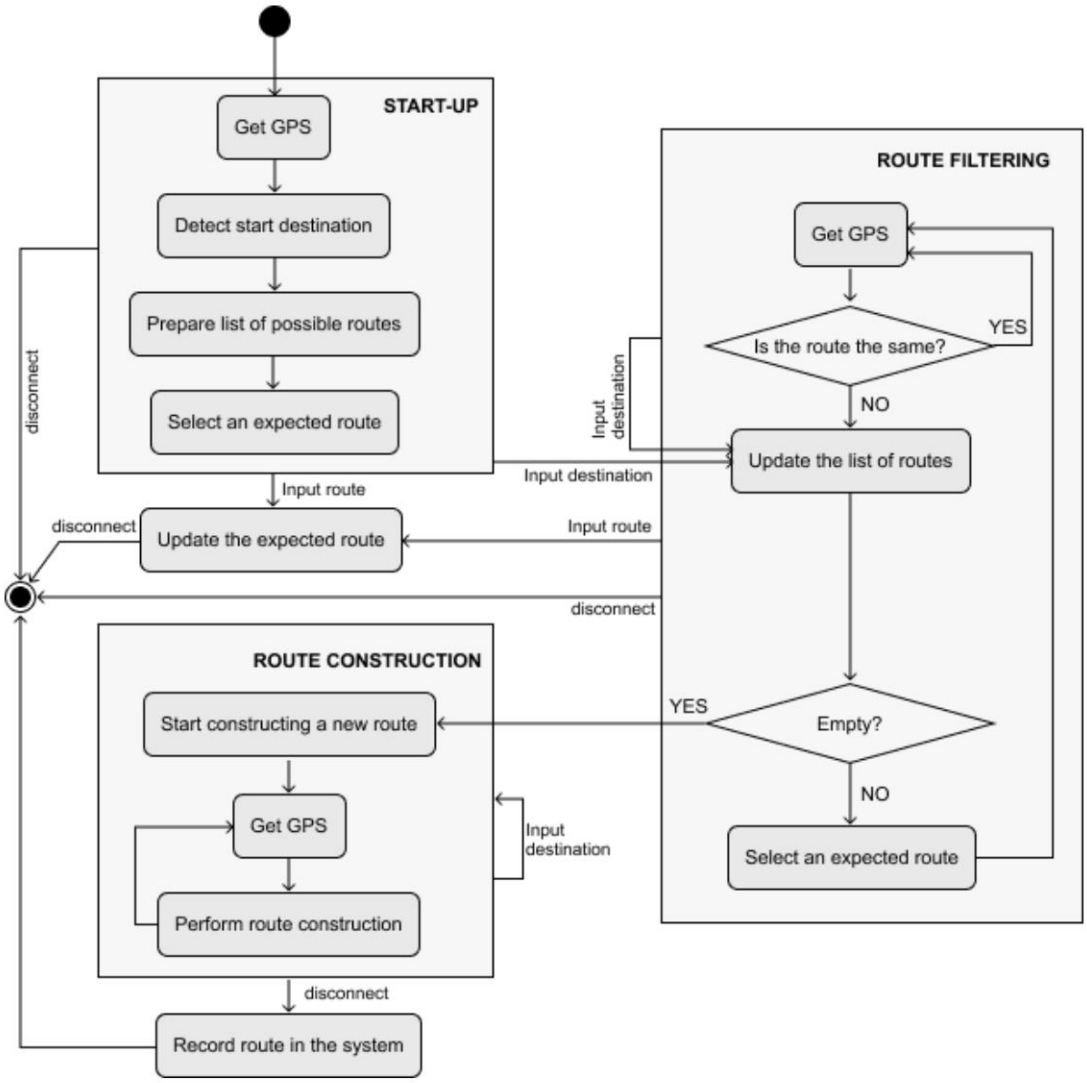


Figure 7. State diagram

Data Structures

Object-oriented programming generally employs data structures such as arrays, lists, stacks, queues, dictionaries, trees, and graphs. These data structures are used to store and manipulate collections of data in an organized and efficient manner. Based on the designed proposal of software to support operations of the DC, several data structures are planned to be used within the context of the classes and their functionalities:

1. **Lists** are useful for storing collections of objects where the order of the elements matters, and duplicates are allowed. In the context of the classes and their functionalities, the Route class may use a list (List<Location>) to maintain waypoints representing the sequence of locations to be visited along a route. The EnvironmentalPerception or Sensors classes could also use lists (List<SensorData>) to store sensor readings or data collected over time.
2. **Queues** follow the First-In-First-Out (FIFO) principle, where the first element added is the first one to be removed. The Navigation class might use a queue (Queue<String>) to manage a queue of destinations waiting to be reached or processed in sequence. The Control class could use a queue (Queue<Double>) to manage a queue of acceleration or braking commands to be executed in sequence.
3. **Dictionaries**, also known as maps, associate keys with values, allowing efficient lookup and retrieval of data based on a unique key. They can be used in various classes, such as the database class, which uses a dictionary (Map<String, Object>) to store and manage data with key-value pairs, enabling operations like inserting, retrieving, updating, and deleting data based on keys. Additionally, the SensorData class could internally use a dictionary

(Map<String, Object>) to store sensor readings with associated metadata or attributes for efficient access and retrieval.

Reference list

Bathla, G., Bhadane, K., Singh, R.K., Kumar, R., Aluvalu, R., Krishnamurthi, R., Kumar, A., Thakur, R.N. & Basheer, S. (2022) 'Autonomous Vehicles and Intelligent Automation: Applications, challenges, and opportunities', Mobile Information Systems, 2022, pp. 1–36. doi:10.1155/2022/7632892.

L'Afflitto, A., İnalhan, G. & Shin, H.-S. (2024) Control of autonomous aerial vehicles: Advances in Autopilot Design for civilian uavs. Cham, Switzerland: Springer.

Leiss, P.J. (2023) The functional components of Autonomous Vehicles, The Functional Components of Autonomous Vehicles. Available at: https://www.robsonforensic.com/articles/autonomous-vehicles-sensors-expert [Accessed: 19 April 2024].

Parekh, D., Poddar, N., Rajpurkar, A., Chahal, M., Kumar, N., Prasad Joshi, G.P., & Cho, W. (2022) 'A review on Autonomous Vehicles: Progress, methods and challenges', Electronics, 11(14), p. 2162. doi:10.3390/electronics11142162.

Pendleton, S., Andersen, H., Du, X., Shen, X., Meghjani, M., Eng, Y.H., Rus, D. & Ang, M.H. (2017) 'Perception, planning, control, and coordination for Autonomous Vehicles', Machines, 5(1), p. 6. doi:10.3390/machines5010006.

Reddy, P. P. (2019) Driverless Car: Software Modelling and Design using Python and Tensorflow.

Sjafrie, H. (2020) Introduction to self-driving vehicle technology. Boca Raton (Fla.): CRC Press Taylor & Francis Group.

Thakurdesai, H.M. & Aghav, J.V. (2020) 'Autonomous cars: Technical challenges and a solution to blind spot', Advances in Intelligent Systems and Computing, pp. 533–547. doi:10.1007/978-981-15-1275-9_44.

Zhao, J., Liang, B. & Chen, Q. (2018) 'The key technology toward the self-driving car', International Journal of Intelligent Unmanned Systems, 6(1), pp. 2–20. doi:10.1108/ijius-08-2017-0008.

Zhou, Z. Q. & Sun, L. (2019) Metamorphic testing of driverless cars. Commun. ACM 62(3): 61–67. DOI: https://doi.org/10.1145/3241979.