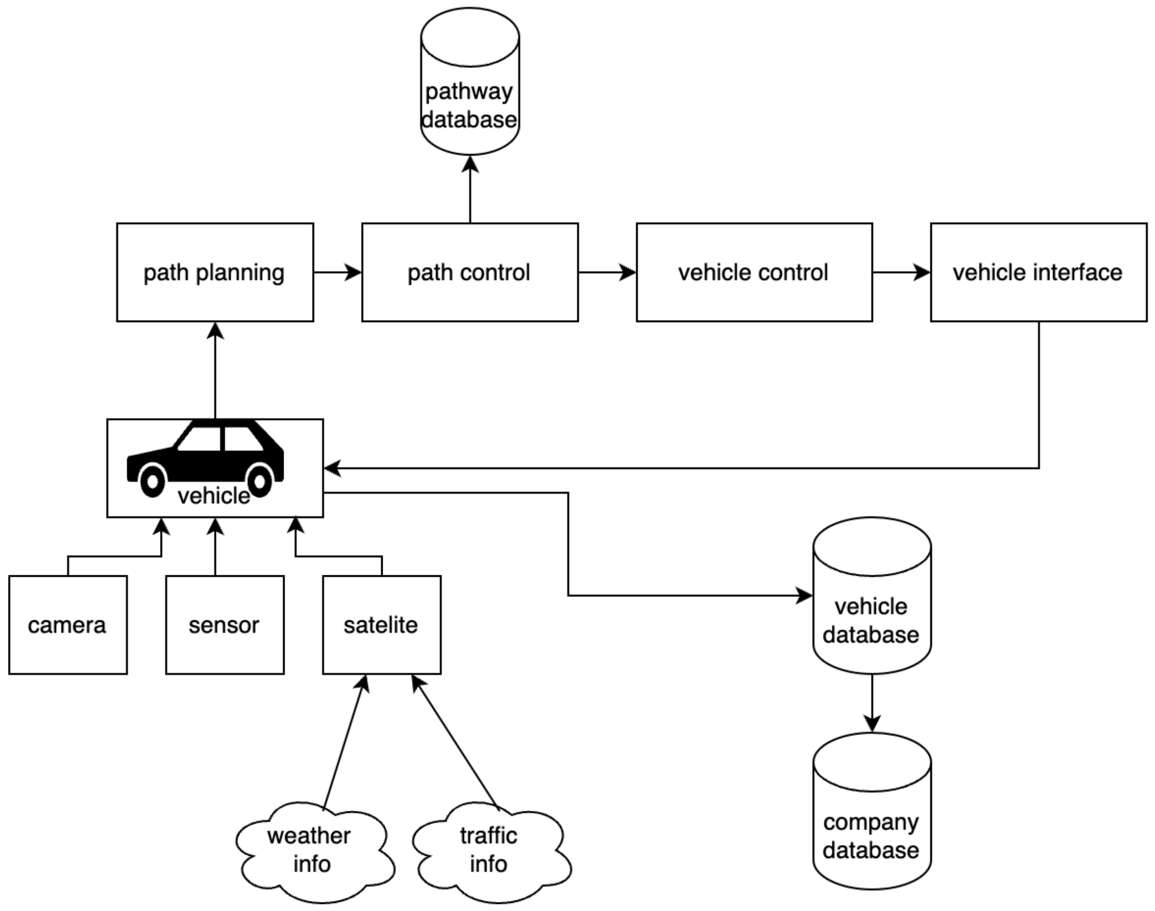
Autonomous Vehicle System

By: Kennedy Kubiak, Evan Tardiff, Trevor Klein

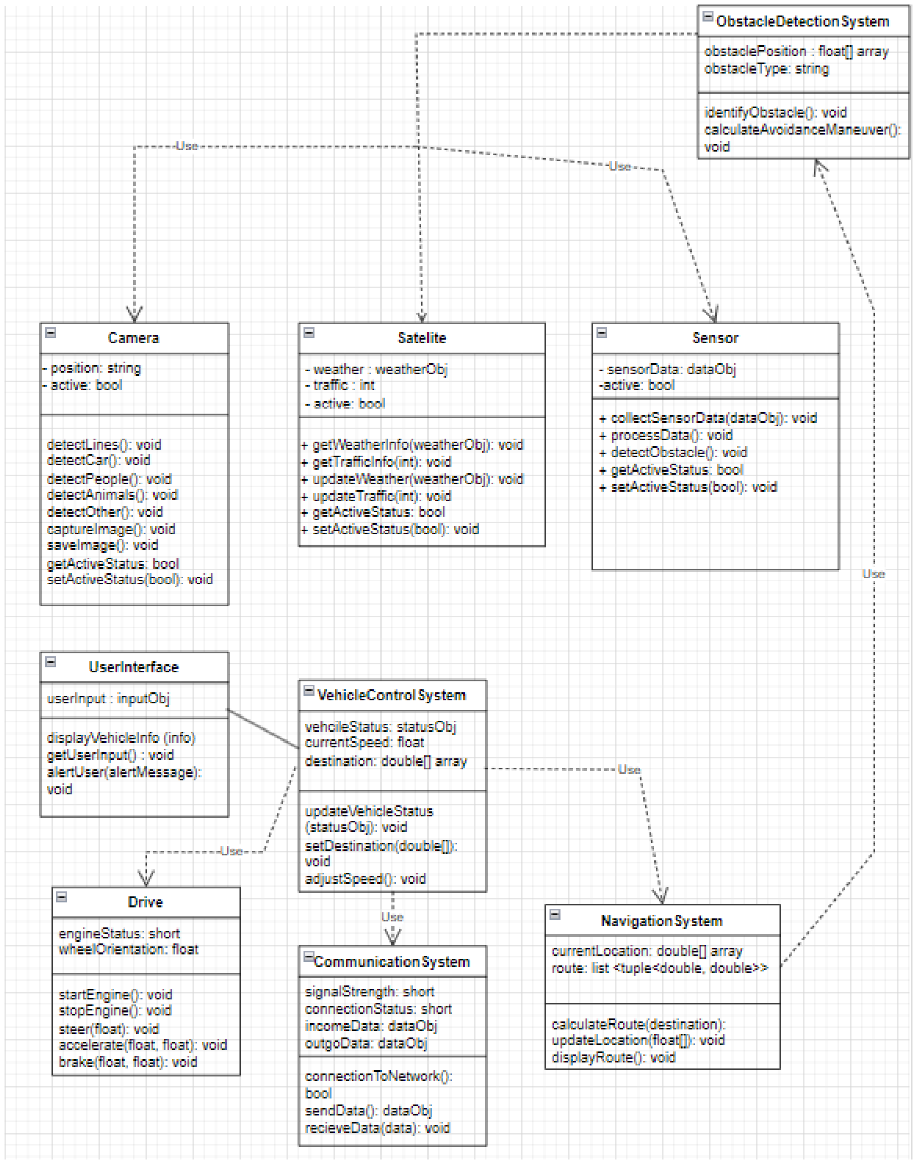
System Description: The Autonomous Vehicle System is needed because it can help increase the safety, efficiency, and reliability of commercial vehicles through enhancing navigation. From the first client interview the system is set to be applied to all current vehicles in the commercial fleet with no plans of expanding as of now. The main services provided by the system will be self-driving between destinations when there are no turns.

Architectural Diagram of all Major Components (SWA Diagram):



Description:
The Software Architecture diagram above represents the major components of the Autonomous Vehicle System and how they interact with each other. The focal component of the diagram is the vehicle. The vehicle receives information input from the camera, sensor, and satellite components which it then uses to make pathway decisions regarding the objects around the vehicle. The satellite component receives traffic and weather information to help make the path decisions relevant to the environment. Within the vehicle, there is a path planning component which uses the input information to make the pathway decisions. These pathway decisions are processed in the path control component. All of these decisions and actions will be stored in the vehicle's pathway database. The path control then interacts with the car control to perform the pathway decisions. This is all informed to the user through the car interface which will tell the user the routes and any necessary information. This all circles back to the vehicle which stores all of the component decisions in a database which is shared to a company database so the company can keep track of their vehicles' performance.

UML Class Diagram:



Description of Classes:

- VehicleControlSystem Class - The class at the center of how the software system operates. It controls the car through use of its own members and through the use of other classes which includes Drive, CommunicationSystem, and NavigationSystem. VehicleControlSystem also has an Association/Aggregation relationship with the UserInterface class.

  VehicleControlSystem has the following attributes:
  - vehicleStatus: holds information about the status of the vehicle in the form of a statusObj which contains information about gas, oil, tire pressure, etc.
  - currentSpeed: represents the speed of the car with a float.
  - destination: using a double array, it stores the coordinates of the destination.

  VehicleControlSystem has the following operations:
  - updateVehicleStatus: Updates the vehicleStatus field.
  - setDestination: takes in a double array representing coordinates as input and sets destination to it.
  - adjustSpeed: Changes the current speed of the vehicle by altering the currentSpeed field.

- UserInterface Class - Represents the UI that the user can interact with and can obtain input from the user. This class has an Association/Aggregation relationship with VehicleControlSystem.

  UserInterface has the following attributes:
  - userInput: Stores the input received by the user.

  UserInterface has the following operations:
  - displayVehicleInfo: Displays information about the vehicle to the user which can include mileage and amount of gas left and other information.
  - Sets the value of userInput to what the user entered in.
  - alertUser: Displays a message to the user. Example could be “Low on gas. Please refuel.”

- Drive Class - Controls the gas pedal and brake pedal functionality of the vehicle. This class is used by VehicleControlSystem to physically get moving, stop moving, and change directions.

  Drive has the following attributes:
  - engineStatus: Using a short, this value represents how good the engine is doing with a higher value being in better condition than a lower value.
  - wheelOrientiation: Using a float, represents the orientation of the wheel with 0.0 meaning that the wheel is straight and the car would drive straight, and 90.0 meaning that the wheel has been turned 90 degrees

        clockwise. Negative values mean that the wheel has been turned counter clockwise.

    Drive has the following operations:
    - startEngine: Turns on the engine of the vehicle.
    - stopEngine: Turns off the engine of the vehicle.
    - steer: Takes in a float value so it can set wheelOrientation to that value.
    - accelerate: Activates the gas pedal. Takes in a float value so it knows what speed to accelerate to and another float to know how fast to do it.
    - brake: Activates the brake pedal. Takes in a float value so it knows what speed to slow down to and another float to know how fast to do it.

- CommunicationSystem Class - This class gets used by VehicleControlSystem and it facilitates outgoing and incoming communication to and from outside networks.

    CommunicationSystem has the following attributes:
    - signalStrength: Uses a short to represent how strong the signal is. The higher the value, the stronger the signal.
    - connectionStatus: Uses a string to display the status of the connection. Examples include “Very Strong” and “Weak” and “No Signal.”
    - incomeData: Holds the information received from an outside network.
    - outgoData: Holds the information that would be sent to an outside network.

    CommunicationSystem has the following operations:
    - connectionToNetwork: Connects to an outside network and returns true if a successful connection occurs and false if not.
    - sendData: Sends out the data stored in outgoData.
    - receiveData: Receives incoming data and stores it in incomeData.

- NavigationSystem Class - This class is used by VehicleControlSystem and uses the class ObstacleDetectionSystem to help the vehicle navigate. It creates a path or route for the vehicle to follow.

    NavigationSystem has the following attributes:
    - currentLocation: Stores the current location of the vehicle in a double array.
    - route: A list of tuples that stores, as doubles, the longitude and latitude coordinates that the vehicle will travel along the route.

    NavigationSystem has the following operations:
    - calculateRoute: takes in the field “destination” from the VehicleControlSystem class and compares it to currentLocation in order to calculate the best route to travel.
    - updateLocation: updates the value of currentLocation.

        - displayRoute: displays the route that was calculated.

- ObstacleDetectionSystem Class - This class is used by NavigationSystem to help it safely navigate through the world by detecting obstacles in the road. It achieves this by using the following classes: Sensor, Satellite, and Camera.

    ObstacleDetectionSystem uses the following attributes:

    - obstaclePosition: Uses a float array to keep track of where an object is. The first index of the array stores how far the object is to the left and right using negative values and positive values respectively. The second index stores how far the object is to the front and the back using positive and negative values respectively.
    - obstacleType: Uses a string to store what the type of object is. For example, an obstacle being a dog can make a big difference to it being a fallen tree.

  ObstacleDetectionSystem has the following operations:

    - identifyObstacle: Identifies what the obstacle is: Is it a person, dog, fallen tree, etc.
    - calculateAvoidanceManeuver: Calculates if a special maneuver to avoid the obstacle is necessary. Also calculates the maneuver if it is necessary. It takes into account what the object is: avoiding a dog or a child requires more caution than avoiding a fallen tree.

- Camera Class - This is one of three classes used by ObstacleDetectionSystem. The Camera class controls how the cameras of the vehicle contribute to detecting obstacles.

    Camera has the following attributes:

    - position: This uses a string so that the program knows which camera this is such as the “rear” camera or “front” camera.
    - active: This boolean will be true if the camera is active and false if it is not active or broken.

    Camera has the following operations:

    - detectLines: This helps the camera detect the lines for lanes on the road so that the vehicle can stay in its line among, know if a line is broken or solid, etc.
    - detectCar: Detects if there is another vehicle nearby and if it is active or just parked.
    - detectPeople: Detects if there is a human nearby.
    - detectAnimals: Detects if there is a non human animal nearby.
    - detectOther: Detects other objects that have lower priority compared to cars, people, and animals, such as a basketball or a rock.
    - captureImage: Saves an image or picture with the camera.

    - saveImage: Saves the image or video taken.
    - getActiveStatus: Returns a boolean based on if the camera is active or not.
    - setActiveStatus: assigns the field active to either true or false to represent if the camera has been activated or not.

- Satellite Class - This is one of three classes used by ObstacleDetectionSystem. The Satellite class facilitates how the vehicle uses satellites to detect obstacles.

  Satellite has the following attributes:
    - weather: This has information about the current weather conditions, such as if it is raining or snowing.
    - traffic: This has information about traffic such as where it is and how bad it is.
    - active: This boolean will be true if the satellite functionality is active and false if it is not active or if something is not working right.

  Satellite has the following operations:
    - getWeatherInfo: Obtains information about the weather and stores it in the weather field.
    - getTrafficInfo: Obtains information about the traffic and stores it in the traffic field.
    - updateWeather: Obtains updated information about the weather and stores it in the weather field.
    - updateTraffic: Obtains updated information about the traffic and stores it in the traffic field.
    - getActiveStatus: Returns a boolean based on if the satellite functionality of the vehicle is active or not.
    - setActiveStatus: Assigns the field active to either true or false to represent if the satellite functionality has been activated or not.

- Sensor Class - This is one of three classes used by ObstacleDetectionSystem.

  Sensor has the following attributes:
    - sensorData: This has information the sensor detects regarding obstacles that are present in or on the road
    - active: This boolean will be true if the sensor functionality is active and false if it is not active or if something is not working right.

  Sensor has the following operations:
    - collectSensorData: Is responsible for taking and storing the data from sensorData in a place that is then accessible to processData
    - processData: Takes the collected sensor data and formats it so that detect obstacle is able to function properly

        - detectObstacle: Uses the processed data to identify possible obstacles that may interfere with the path set by the navigation system
        - getActiveStatus: Returns a boolean based on if the sensor functionality of the vehicle is active or not.
        - setActiveStatus: Assigns the field active to either true or false to represent if the sensor functionality has been activated or not.

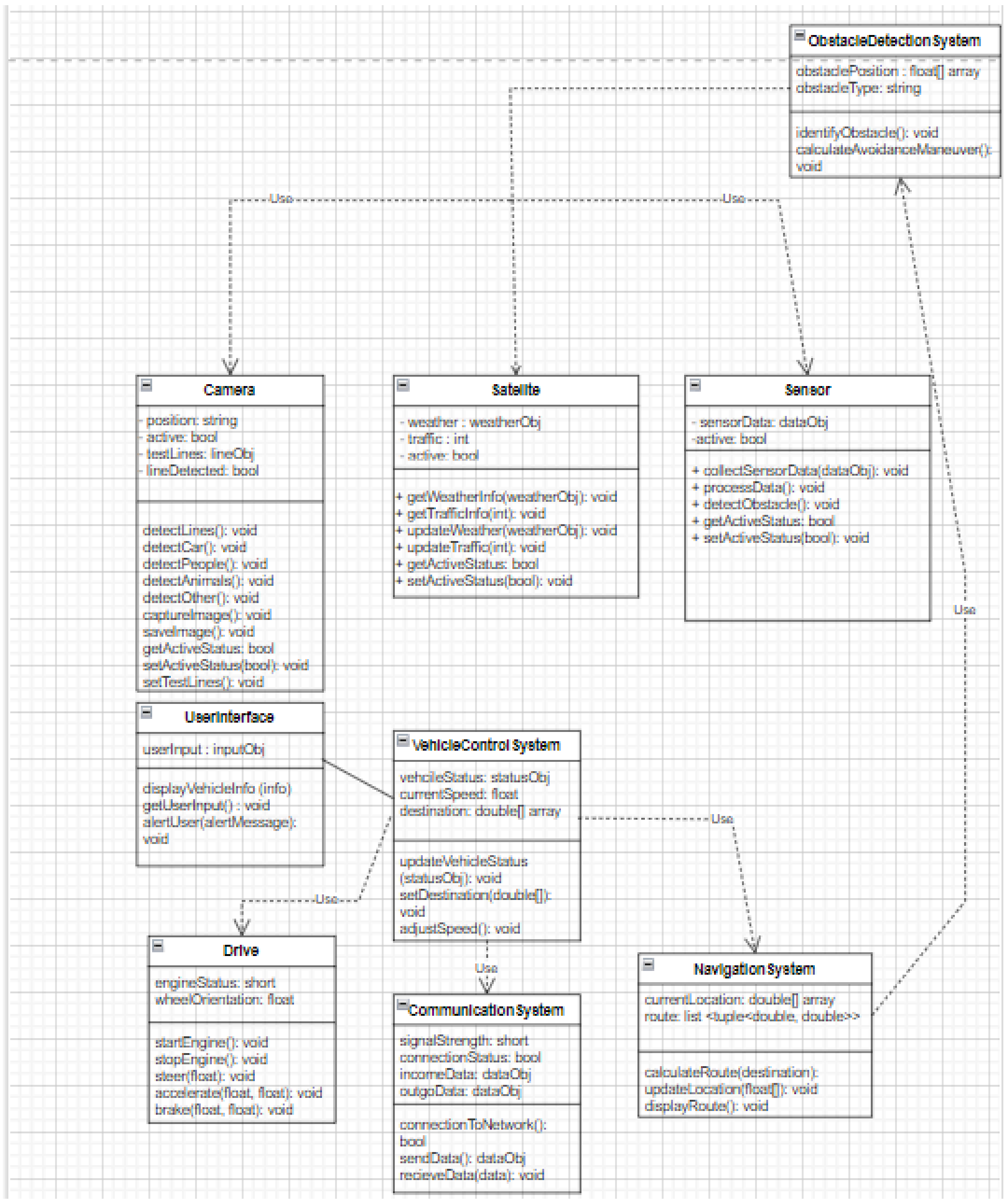
Development Plan and Timeline:
For the development of the Autonomous Vehicle System, we plan to use both an iterative and waterfall approach. The development process will span the course of two years and we all plan to be equally involved in each step of the process. As we already have the project requirements given to us, we can begin with designing the prototypes. This process should take around four months. Then we will spend six months designing the hardware such as the cameras, sensors, and satellites of the vehicle. After that, we will be able to begin the software development phase. This will take us six months to create the software such as the navigation algorithms and the user interface. After everything is completed, we can begin testing. We will test for around eight months to make sure it is reliable and safe through all conditions and cases. During testing, we will evaluate the system and identify any areas for improvement and update it as needed.

Updated UML Diagram
- Changes to the Camera Class:
    Added the following attributes to the Camera Class:
        - testLines: This object holds the lines being used to test our detectLines() function.
        - lineDetected: This boolean will be true if our detectLines() function is able to detect the lines and false if it can not.

    Added the following operations to the Camera Class:
        - setTestLines: Updates the testLines variable that will be used to test our detectLines function

- Changes to the CommunicationSystem Class
    - Modified the following attributes in the CommunicationSystem Class:
        - connectionStatus: short -> connectionStatus bool. This change came as a result of both signalStrength and connectionStatus being used for the same purpose previously. By changing connectionStatus to a bool we are able to

differentiate connectionStatus from signalStrength and make use of this change in our testing



Test Plan

<u>Camera</u> - detectLines() : void

- **Unit -**

testLinesUnit(Obj line_image, bool expected){

Camera cam

```
    cam.setTestLines(line_image)
    cam.detectLines()
    if(cam.lineDetected == expected){
        return “pass”
    }
    return “fail”
```

We will use this multiple times to test different types of lines such as solid, broken, curved, white, yellow, etc. We are plugging in a picture for each scenario in for line_image to test each separate case along with what the expected output should be for the test. This is partitional since we are extrapolating the results. If the returned value is “pass” then the detectLines() function returns the expected value and works for the given image.

- **Functional** -

```
testLinesFunctional(Obj obs_image, bool expected){
    ObstacleDetectionSystem obs
    obstacle = obs.identifyObstacle()
    if(obstacle == “Line”){
        obs.setTestLines(obs_image)
        obs.detectLines()
        if(obs.lineDetected == expected){
            return “pass”
        }
        return “fail”
    if(expected == True) {
        return “fail
    }
    return “pass”
```

In this test method, we are testing how detectLines() works with the ObstacleDetectionSystem class. Obstacle images could be input coming from either Camera, Satellite, or Sensor. It could be lines, or maybe not, that is where the testing happens. If it is categorized as “Line” by the identifyObstacle() method, then it goes through to be checked by detectLines() which is what we are testing, and it returns pass or fail depending on the image and expected result. If the expected was supposed to be a line then the second if statement will cause the test to also return fail, this time checking to see if the identifyObstacle() function is working. This is partition testing since we are testing with different inputs and extrapolating the results to those of similar categories.

- **System** - A user is driving around and the ObstacleDetectionSystem is able to use the Camera class and the vehicle's cameras to help the user identify obstacles on the road. Specific example is when the user is trying to turn and the cameras detect the lines of the road to help the user stay in the lane and not get beyond the lines.

<u>CommunicationSystem</u> - connectionToNetwork() : boolean

- **Unit** -

```
checkConnectionToNetworkUnit(bool expected){
    CommunicationSystem c
    c.signalStrength = -1
    c.connectionStatus = null
    c.incomeData = data
    c.outgoData = data
    c.connectionStatus = c.connectionToNetwork()
    if(c.connectionStatus == expected){
        return "pass"
    }
    return "fail"
}
```

To check connectionToNetwork(), we will pass 'true' and 'false' boolean values into the expected variable. In the unit test, we call connectionToNetwork() to our CommunicationSystem object that will update the connection to the network for the communication system. If the connection status of the object is equal to our expected value, it will return 'pass' and if not it will return 'fail'. We used a brute force approach as we used all possible cases. If our unit test passes for both possibilities, we know that the connectionToNetwork() method works correctly.

- **Functional** -

```
checkConnectionToNetworkFunctional(Obj testVehicleStatus, bool expected){
    VehicleControlSystem vcs
    vcs.updateVehicleStatus(testVehicleStatus)
    vcs.connectionStatus = vcs.connectionToNetwork()
```

```
        if(vcs.connectionStatus == expected){
                return “pass”
        }
        return “fail”
}
```

In this test method we are making a new object of the VehicleControlSystem class. We are then calling the updateVehicleStatus method and passing our test object. We then call the connectionToNetwork method that we are trying to test and set the connectionStatus variable equal to its return. We then check this returned value against our expected value. If the two are equal the functional test passes, if they are not equal the functional test fails. For this test we are partitioning the tests by the objects that we are passing. For example we will pass an object that changes the signalStrength, then if it works extrapolate the result to apply to all instances of updating signalStrength.

- **System** - The user should be able to get in the car and turn it on. After turning on the car, they should be able to access a user interface. The connection to network should be displayed on the user interface showing the signal strength. If the interface shows the connection to network as expected, we know that our system works as expected. If no network connection appears, it did not work as expected and our test failed.