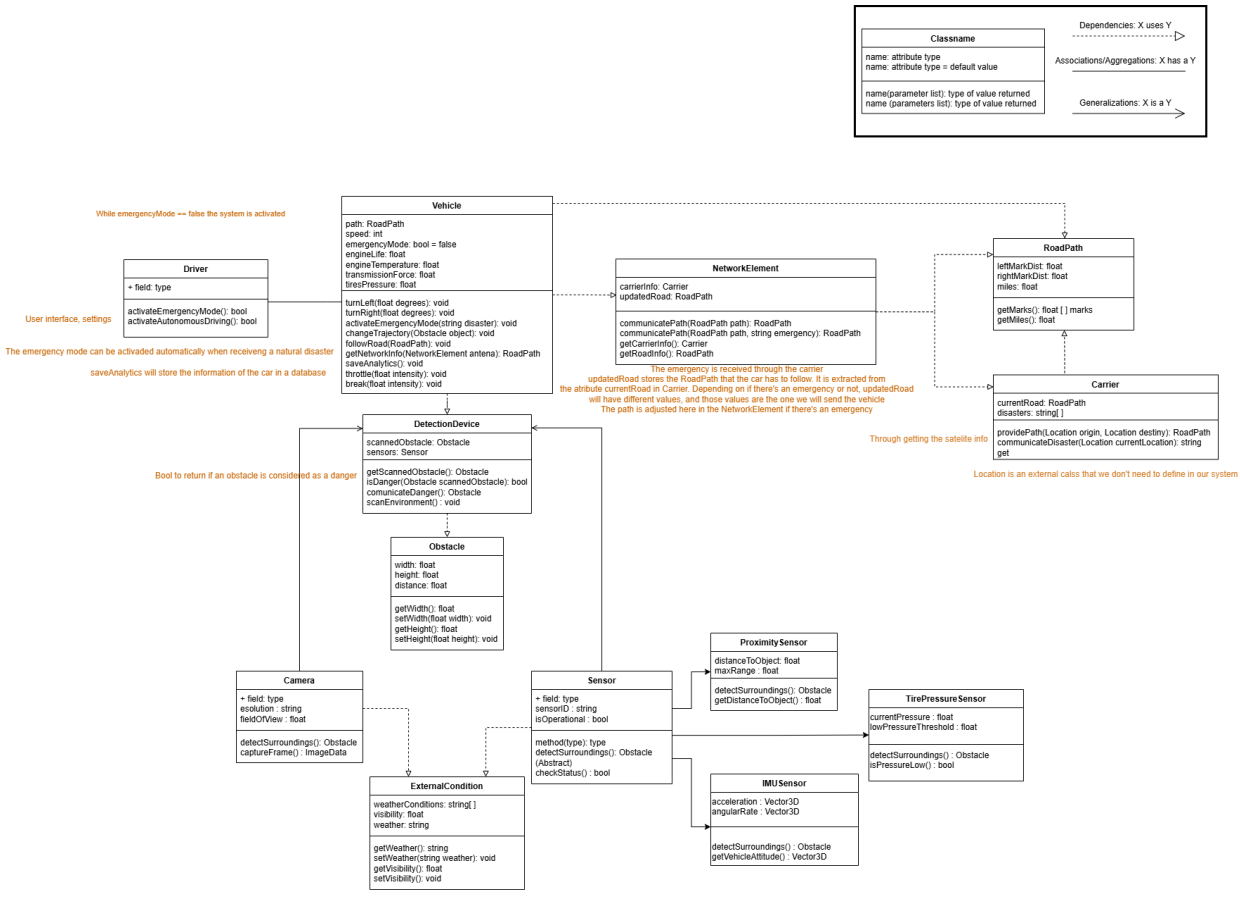
# Autonomous Vehicle System

Anwar, John, Daniel, Gabe, Rodolfo, Joseph

CS 250: Introduction to Software Systems

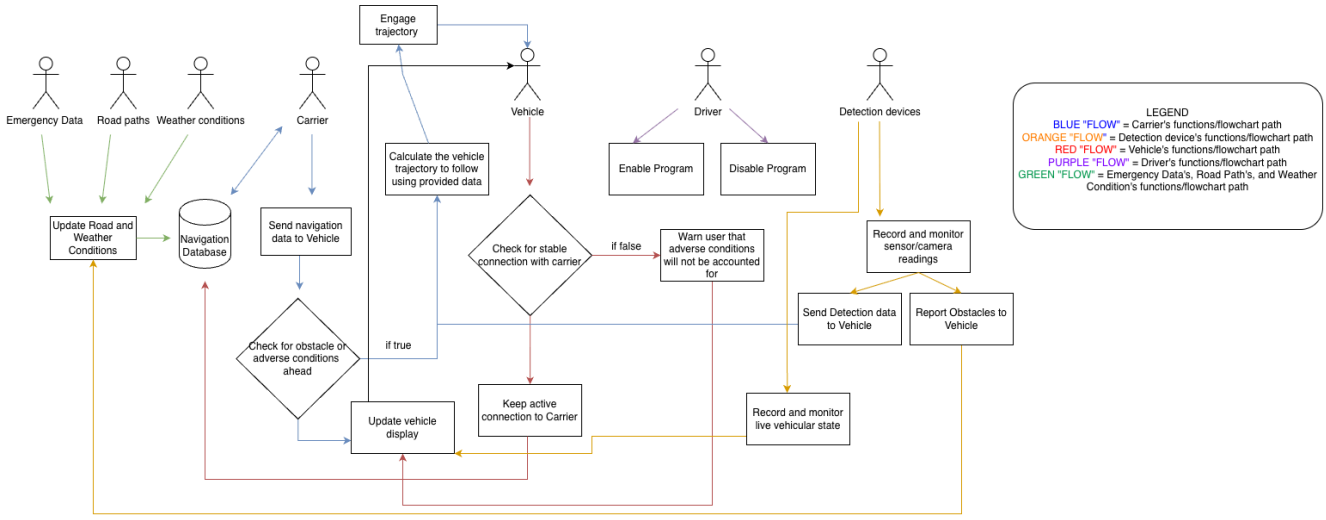
Professor Umut Can Cabuk

Updated UML Class Diagram:




We have made some minor updates to the UML Class Diagram, adding an intensity parameter to the throttle and brake function in Vehicle as well as deleting the relationship existing between Vehicle and ExternalCondition.

Updated SWA Diagram:




We have updated the legend to include a color-coordinated flow for each actor. This way, we can follow what functions they have access to and use in the functionality of the code. We believe that it will help to quickly identify “flows” or paths that can be taken.

**TEST SET #1:**
Targets and tests the DetectionDevice class and its subfunctions in assessing and evaluating the environment around the vehicle.

**Unit:**

```
// First we declare the necessary constants
const dangerDistance = 10.0
const safeWidth = 0.25
const safeHeight = 0.25

// Obstacles that are at the exact danger distance and different measures
Obstacle obs1 = Obstacle(dangerDistance, safeWidth, safeHeight)
Obstacle obs2 = Obstacle(dangerDistance, safeWidth, safeHeight + 0.1)
Obstacle obs3 = Obstacle(dangerDistance, safeWidth + 0.1, safeHeight)
Obstacle obs4 = Obstacle(dangerDistance, safeWidth + 0.1, safeHeight + 0.1)
Obstacle obs5 = Obstacle(dangerDistance, safeWidth, safeHeight - 0.1)
Obstacle obs6 = Obstacle(dangerDistance, safeWidth - 0.1, safeHeight)
Obstacle obs7 = Obstacle(dangerDistance, safeWidth - 0.1, safeHeight - 0.1)
Obstacle obs8 = Obstacle(dangerDistance, safeWidth + 0.1, safeHeight - 0.1)
Obstacle obs9 = Obstacle(dangerDistance, safeWidth - 0.1, safeHeight + 0.1)

// Obstacles that are closer than the danger distance and different measures
Obstacle obs10 = Obstacle(dangerDistance - 0.1, safeWidth, safeHeight)
Obstacle obs11 = Obstacle(dangerDistance - 0.1, safeWidth, safeHeight + 0.1)
Obstacle obs12 = Obstacle(dangerDistance - 0.1, safeWidth + 0.1, safeHeight)
```

```
Obstacle obs13 = Obstacle(dangerDistance - 0.1, safeWidth + 0.1, safeHeight + 0.1)
Obstacle obs14 = Obstacle(dangerDistance - 0.1, safeWidth, safeHeight - 0.1)
Obstacle obs15 = Obstacle(dangerDistance - 0.1, safeWidth - 0.1, safeHeight)
Obstacle obs16 = Obstacle(dangerDistance - 0.1, safeWidth - 0.1, safeHeight - 0.1)
Obstacle obs17 = Obstacle(dangerDistance - 0.1, safeWidth + 0.1, safeHeight - 0.1)
Obstacle obs18 = Obstacle(dangerDistance - 0.1, safeWidth - 0.1, safeHeight + 0.1)

// Obstacles that are further than the danger distance and different measures
Obstacle obs19 = Obstacle(dangerDistance + 0.1, safeWidth, safeHeight)
Obstacle obs20 = Obstacle(dangerDistance + 0.1, safeWidth, safeHeight + 0.1)
Obstacle obs21 = Obstacle(dangerDistance + 0.1, safeWidth + 0.1, safeHeight)
Obstacle obs22 = Obstacle(dangerDistance + 0.1, safeWidth + 0.1, safeHeight + 0.1)
Obstacle obs23 = Obstacle(dangerDistance + 0.1, safeWidth, safeHeight - 0.1)
Obstacle obs24 = Obstacle(dangerDistance + 0.1, safeWidth - 0.1, safeHeight)
Obstacle obs25 = Obstacle(dangerDistance + 0.1, safeWidth - 0.1, safeHeight - 0.1)
Obstacle obs26 = Obstacle(dangerDistance + 0.1, safeWidth + 0.1, safeHeight - 0.1)
Obstacle obs27 = Obstacle(dangerDistance + 0.1, safeWidth - 0.1, safeHeight + 0.1)


testsVector = [obs*]    // Add every obstacle to testsVector
resultsVector = [isDanger(test) for test in testsVector]

/*Expected output for obstacles that are at the exact danger distance and different
measures*/
expectedResults1 = [0, 1, 1, 1, 0, 0, 0, 1, 1]

/*Expected output for obstacles that are closer than the danger distance and different
measures*/
expectedResults2 = [0, 1, 1, 1, 0, 0, 0, 1, 1]

/*Expected output for obstacles that are further than the danger distance and different
measures*/
expectedResults3 = [0, 0, 0, 0, 0, 0, 0, 0, 0]

// Now we combine all the expected results in a single list
expectedResultsVector = expectedResults1 + expectedResults2 + expectedResults3

If resultsVector == expectedResultsVector:
        return PASS    // The test has passed
else:
        // Print the vectors to see which tests failed
        print(resultsVector)
        print(expectedResultsVector)
        return FAIL      // The test has failed
```

**Explanation**:
We test the isDanger function considering if a specified obstacle received as a parameter is considered a **danger** to the vehicle or not. For that, we try with different types of obstacles from the **Obstacle** class (width, height, distance). If the obstacle's width or height is less or equal than 0.25 meters we do not consider the obstacle as a danger to the vehicle. If the obstacle is strictly more than 10.0 meters away no matter its width or height, it will not be considered as a danger either.

In other words, a **danger** is an obstacle with:
**distance <= 10 && (weight > 0.25 || height > 0.25)**

First, we build all possible critical combinations of obstacles. Then, we add them all to a vector called 'testsVector'. After that, we build our 'expectedResultsVector' vector and we apply the function we want to test (inDanger(Obstacle)) to every obstacle in the test vector. Lastly, if the vectors are the same we conclude that the test is successful, otherwise, we print both vectors to see which results are different and we conclude that the test is unsuccessful.

**Integration:**

```
ProximitySensor()
maxRange = 10
obstacle f
obstacle t
distanceToObject(t) = 11
distanceToObject(f) = 7

If detectSurroundings() == f, t
        return PASS
else
        return FAIL
DetectionDevice()
If isDanger(f) = false && isDanger(t) = true
        return PASS
else
        return FAIL
```

**Explanation**:
The integration test makes sure that the ProximitySensor class works when detecting and assessing the environment around the vehicle and that DetectionDevice can communicate and receive important information. Here in this example, we make sure

that the detectSurroundings() method correctly returns obstacles surrounding a vehicle when they're detected to be closer than the maxRange distance. Not passing this test demonstrates a serious risk for the driver since it jeopardizes their safety and more tests should be conducted on the whole class until correct functionality is achieved.

**System:**

```
// 1. Setup Environment
// Define constants for the danger criteria (matching Unit Test)
const DANGER_DISTANCE = 10.0
const DANGER_WIDTH = 0.5
const DANGER_HEIGHT = 0.5

// 2. Create and Configure Components
// Instantiate the core system components
Vehicle car = new Vehicle()
Driver driver = new Driver(car) // Assuming Driver holds a reference to the Vehicle
DetectionDevice detector = new DetectionDevice()
ProximitySensor sensor = new ProximitySensor()

// Connect components (as implied by the diagram's dependencies)
detector.addSensor(sensor)
car.setDetectionDevice(detector)
car.speed = 30.0 // Set initial speed > 0 for emergency mode to be relevant

// 3. Simulate the Input (Obstacle Detection)
// Create an obstacle that MUST be considered a danger:
// Closer than DANGER_DISTANCE and larger than DANGER_WIDTH/HEIGHT
Obstacle dangerousObs = new Obstacle()
    distance: DANGER_DISTANCE - 1.0, // e.g., 9.0 meters away
    width: DANGER_WIDTH + 1.0,      // e.g., 1.5 meters wide
    height: DANGER_HEIGHT + 1.0     // e.g., 1.5 meters high
)

// Simulate the ProximitySensor detecting the dangerous obstacle
sensor.setDetectedObstacles([dangerousObs])

// 4. Execute the System Flow
// The Driver initiates the periodic safety check.
driver.checkEmergencyMode() // This should call car.activationEmergencyMode()

// 5. Verification (Expected Outcome)
// Check if the Vehicle's emergency state has been activated.
if car.isEmergencyModeActive() == true:
```

```
    return PASS
else:
    // This indicates a total failure in the safety chain
    print("System failed to activate emergency mode for a dangerous obstacle.")
    return FAIL
```

**Explanation**:
The critical system components, including the vehicle, driver, detection device, and proximity sensor, are initialized and connected. The system test is designed to validate the core safety functionality of the vehicle; specifically, the mechanism for activating emergency mode upon detection of a hazardous object. This test confirms that the complete chain of dependencies, from the lowest level sensor input to the highest level vehicle logic, works as intended. Failing this test indicates a critical system flaw, as it means the vehicle is incapable of automatically reacting to an immediate emergency, which is the primary goal of the system's design.

**TEST SET #2:**
Targets and tests vehicle actuation and control with an emphasis on the throttle(intensity) and brake(intensity) functions.

**Unit:**

```
// Define maximum limits for control for simulation reference
const MAX_THROTTLE = 1.0
const MAX_BRAKE = 1.0
const MAX_SPEED = 100.0 // kph

// 1. Setup Vehicle
Vehicle testCar = new Vehicle()

// 2. Test throttle function
// A. Full Throttle Check
testCar.throttle(MAX_THROTTLE)
// Expected: currentSpeed increases significantly, throttleLevel is max
expectedSpeedA = testCar.currentSpeed + 5.0 // Simulate a change
expectedThrottleA = MAX_THROTTLE

if testCar.getThrottleLevel() != expectedThrottleA:
    return FAIL // Throttle state not set correctly

// B. Partial Throttle Check
testCar.throttle(0.5)
// Expected: currentSpeed increases, but less than MAX_THROTTLE
expectedThrottleB = 0.5
expectedSpeedB = testCar.currentSpeed + 2.5 // Simulate a change
```

```
if testCar.getThrottleLevel() != expectedThrottleB:
    return FAIL // Throttle state not set correctly

// 3. Test brake function
testCar.currentSpeed = 50.0 // Set a measurable initial speed

// C. Full Brake Check
testCar.brake(MAX_BRAKE)
// Expected: currentSpeed decreases significantly, throttle is zero
expectedSpeedC = 0.0 // Simulate a full stop
expectedThrottleC = 0.0 // Brake implies no throttle
expectedBrakeC = MAX_BRAKE

if testCar.getCurrentSpeed() != expectedSpeedC or testCar.getBrakeLevel() !=
expectedBrakeC:
    return FAIL // Brake application or speed change failed

// D. Partial Brake Check (simulate slow-down)
testCar.currentSpeed = 50.0 // Reset speed
testCar.brake(0.3)
// Expected: currentSpeed decreases moderately
expectedSpeedD = 35.0 // Simulate slow-down
expectedBrakeD = 0.3

if testCar.getCurrentSpeed() != expectedSpeedD or testCar.getBrakeLevel() !=
expectedBrakeD:
    return FAIL // Partial brake or speed change failed

// 4. Combined check (e.g., stopping throttle when braking)
testCar.throttle(0.8) // Set throttle
testCar.brake(0.5) // Brake should override/clear throttle
expectedThrottleE = 0.0 // Throttle should drop to 0
expectedBrakeE = 0.5
if testCar.getThrottleLevel() != expectedThrottleE:
    return FAIL // Brake did not correctly clear throttle

return PASS
```

**Explanation**:
This unit test validates the internal logic of the Vehicle class's core control methods, throttle(intensity) and brake(intensity). It ensures that the functions can correctly execute the following tasks: setting the internal control state (throttleLevel, brakeLevel) according to the intensity parameter, applying the expected effect on the vehicle's state

(currentSpeed) for both partial and full inputs, and handling basic control conflict, specifically that applying the brake should implicitly or explicitly set the throttleLevel to zero. This directly tests the implementation details introduced in the updated UML diagram.

**Integration:**

```
// 1. Setup Components
Vehicle car = new Vehicle()
Driver driver = new Driver(car)

// 2. Simulate Driver Input (Acceleration)
const ACCEL_INTENSITY = 0.7
// Assuming the Driver class has a method to request acceleration
driver.requestAcceleration(ACCEL_INTENSITY) // This should call car.throttle(intensity)

// 3. Verification
// Check if the Vehicle received and processed the correct command
expectedSpeedIncrease = 3.5 // Simulated speed increase for intensity 0.7

// Check Vehicle's internal state
if car.getThrottleLevel() == ACCEL_INTENSITY:
   // Check Vehicle's output state
   if car.getCurrentSpeed() == expectedSpeedIncrease:
      return PASS // Driver input correctly actuated the Vehicle
   else:
      print("Vehicle speed change did not match expected output.")
      return FAIL
else:
   print("Vehicle did not receive or process the correct throttle intensity from Driver.")
   return FAIL
```

**Explanation**:
This integration test ensures that the Driver class can successfully communicate a control command (like acceleration) to the Vehicle class and that the Vehicle correctly executes the corresponding throttle(intensity) function. It verifies the interface and communication between two dependent classes, ensuring the high-level decision (Driver) correctly translates into the low-level action (Vehicle control).

**System:**

```
// 1. Setup Environment
// (Constants and Components are the same as Test Set #1, but focus is on normal control)
Vehicle car = new Vehicle()
Driver driver = new Driver(car)
```

```
ControlSystem controller = new ControlSystem(car) // Assuming a control system for non-emergency logic
car.currentSpeed = 20.0
const TARGET_SPEED = 50.0

// 2. Simulate Input (ExternalCondition changes, or a pathfinding update)
// The ControlSystem decides the vehicle needs to accelerate to 50.0 kph
controller.setTargetSpeed(TARGET_SPEED)

// 3. Execute the System Flow
// The ControlSystem calculates the necessary control input (e.g., 0.6 throttle)
// to reach the target speed and apply it.
const CALCULATED_INTENSITY = 0.6 // The system decides this based on the speed delta
controller.executeSpeedControl(CALCULATED_INTENSITY) // This calls car.throttle(0.6)

// 4. Verification (Expected Outcome)
// Check if the Vehicle's control state has been correctly updated by the ControlSystem.
expectedSpeedAfterThrottle = 23.0 // Simulated increase for intensity 0.6

if car.getThrottleLevel() == CALCULATED_INTENSITY:
    if car.getCurrentSpeed() == expectedSpeedAfterThrottle:
        return PASS
    else:
        print("Vehicle failed to update speed after ControlSystem input.")
        return FAIL
else:
    // This indicates a failure in the normal control loop (ControlSystem -> Vehicle)
    print("System failed to correctly apply the calculated throttle intensity.")
    return FAIL
```

**Explanation**:
The system test verifies the normal operational control loop of the Autonomous Vehicle. It validates that the high-level ControlSystem can analyze a goal (e.g., reach TARGET_SPEED), calculate the appropriate control intensity (e.g., CALCULATED_INTENSITY), and successfully command the Vehicle to execute the corresponding action (throttle(intensity)). This ensures the non-emergency, everyday driving functionality, which relies on the throttle and brake updates, is fully functional.