SARL ELEVATOR PROJECT: PART 2

Dylan Rock – s3294558
Joshua Beale - s3413194
Matthew Mcnally - s3488557
Joshua Richards - s3490925

Contents

3. Elevator Simulation Development

3.1 Our goals

The aim of this project was to take the current state of the elevator simulator and allow a user to host the simulation on a client-server system, have users connect to this server and test their own agents within the SARL language.
This task we looked at and assessed that to make attain this goal we would need to develop the following tasks:

- Take separate the events that are passed directly to the simulation and allow them to be transmitted to the simulation across a TCP IP connection for remote access
- Develop a Network controller to allow the transmission of these events to and from the elevator simulator

3.2 Overview

Everything within the current state of the simulator was developed to be run on the same machine, assessing what parts of the simulation needed to change when initialising the server meant that we needed to separate areas of the code that needed to be transmitted.

The Wrapper controller was similar to what was already implemented in the original elevator simulator though this controller now has a network interface tied to it for TCP communication. We used the same controller interface that was present in the simple and meta controller but most of the logic that they had implemented for event handling was doing nothing because of the way that we now had to handle events, now we use an observer and listener set to detect when an event has been sent from the client, the simulation then performs this action, /then responds with the appropriate message back to the client to ensure that their representation of the current state of the simulation is correct.

The changes we have made to events and the transmittable interface that we have been working on. The initial build of the elevator simulation processed events at a certain time that was pre built at the start of every simulation, for our network connection to be possible we had to change the processing of an event to also notify the client in order to keep concurrency.

The handling of events need to change as well, the events that were being executed on the server side now generated a transmittable object that the client would see to verify that actions were taking place on the server side, this was achieved by making each of the events that are being accessed having their own unique identifier and description that our listener could grab and send a message to the client. This message was constructed by taking the identifying aspects of the event and constructs a JSON message that is converted to a string and send to the client. This process also has it's own error handling in that it ensures that the messages that is to be transmitted is correctly associated with an action to be taken via the ID and description

When sending information to the client the use of sensors were involved in collecting the actions that were performed by the simulator when passing through the event queue, these sensors were attached to all events that can be executed by the simulation during a normal run. When an event has been performed the sensors will add a percept to the event queue at the current simulation time, the percepts acts like an event in that it is processed and passed through the event transmitter, but it has no action tied to it, the reason for this event spawning is so that rather than having the events being passed back and forth the client only receives the percept, as a form of acknowledgement that the server has performed an action that they have passed.

4. Wrapper elements

Source: **Elevator-Sim/src/app/io/sarl/wrapper/action**

4.1 Actions

There are three actions that we have developed for the use within the wrapper, they are the **SendCarAction, ChangeNextDirection** and the **ErrorAction.** These actions dictate the movement that the **Cars** will perform based on the car and floor requests of the car riders. Contained within the **Action** source is also a **ListenerThread** which is the receiver of the actions and performs the functionality tied to the **Actions.**

4.1.1. Send Car Action

Package: **Elevator-Sim/src/app/io/sarl/wrapper/action/SendCarAction.Java**

The **SendCarAction** is used to respond to a **CarRequest** made by people in the building on a floor. Once a car request is made the **Floor** and **Direction** are sent to the car, and based on the Cars current location the car will move in the direction that is required to move to the **Floor** of the request. There is an error check put in place to check whether the floor that has been requested and the car that has been requested is valid within the **Model.**

4.1.2. Send Car Action

Package: **Elevator-Sim/src/app/io/sarl/wrapper/action/ChangeNextDirectionAction.Java**

ChangeNextDirection functions similarly to **SendCarAction** as it takes the takes the **Direction** and **Floor** that has been requested and moves the **Car** to the requested floor. The difference is that the **ChangeCarDirection** allows the **Car** to change **Direction** between requests that are going in different directions.

4.1.3. Error Action

Package: **Elevator-Sim/src/app/io/sarl/wrapper/action/ErrorAction.Java**

If a client sends an **Action** to the elevator simulator that is not valid for any reason, the **ErrorAction** sends back this response in order to inform the user that there has been a failed **Action** attempted

4.1.4. Action Processing

Package: **Elevator-Sim/src/app/io/sarl/wrapper/action/Action.Java**

Actions are passed through the **EventQueue** (refer to x.x.x.) and has an **ActionID** and **description** that are associated with it. These actions are processed immediately so when they are at the front of the **EventQueue** the current time is returned and the **Action** is performed.
There are three states in which an **Action** can be in once they have been sent from the client. The **Completed** state informs the user that the **Action** is valid and has been successfully executed. The **inProgress** state means that the action is valid and is currently being performed by the elevator simulator. The **Failed** state is returned when the **action** is not valid and therefore not performed.

4.2. Events, Percepts, Sensors and Transmission

Package: **Elevator-Sim/src/app/io/sarl/wrapper/Event**

Events in the original simulation events were passed through the event queue in a chronological order, over the course of our development we have made these events transmittable across a network connection through the use of JSON messages, identifiable amongst other events and have used percepts to give feedback to a client connected to the simulator.

4.2.1.Event Identifiers (Refer to 4.5 for detailed information about each message)

Each **event** performs a different function within the original elevator simulator and in our changed version of the simulator. For our Server-Client model the need to transmit these events and have them discernable from one another was important for the client. We have created an ID that is tied to each of the events that the simulator completes.

4.2.2. Event Transmission

Source: **Elevator-Sim/src/app/io/sarl/wrapper/Event/EventTransmitter.java**

The **EventTransmitter** implements the **Listener** that the **EventQueue** creates. Once receiving an **Event** from the **EventQueue** the transmitter gets the **Name, Time** and **ID** of this event, called a **Description,** and then construct a **JSON** message to be sent to the client through the **socket** and network connection. There is an exception that is is thrown if at any stage the event cannot be transmitted over the network interface

4.2.3. Percepts

Source: **Elevator-Sim/src/app/io/sarl/wrapper/Event/Percept.java**

Event responses that are to be transmitted through the use of **JSON** messages are not sent in their original state. For centralisation purposes each **event** processed through the **EventQueue** generates a percept, which is simply a timestamp and **Description** attached to that **Event.**

4.2.4. Sensors

The **Sensors** are responsible for the creation of a **percept**, the two sensors that are have been created for this project are the **FloorRequestSensor** and **WorldModelSensor.** These **Sensors** receive the information of the object that they are attached to and create a **Percept** that is then sent to the client.

4.2.4.1. Floor Request Sensor

Source: **Elevator-Sim/src/app/io/sarl/wrapper/Event/FloorRequestSensor.java**

The **FloorRequestSensor** is in control of sending a percept when a **Person** in a **Car** request to move to a given **Floor,** When this event is triggered the **FloorRequestSensor** gets the **CarID** and the **FloorRequest** that has been made and sends that information back to the client in the form of a **JSON** message.

4.2.4.2. World Model Sensor

Source: **Elevator-Sim/src/app/io/sarl/wrapper/Event/WorldModelSensor.java**

The **WorldModelSensor** acts much in the same way that the **FloorRequestSensor** does in that it has a listener that waits for the **Model** of the building to change. This occurs in two stages of the Simulator, the initial construcion of the building within the Simulation and periodically if the simulations building **Model** changes. The information that is sent to the client as a percept includes the number of **Floors** that have been created, the number and details of **Car** objects that are created including their **capacity, ID, servicedFloors** and **currentHeight.**

4.3. Network Helper

Package: **Elevator-Sim/src/app/io/sarl/wrapper/NetworkHelper.java**

The network connection begins with developing a **Socket** interface, the **Socket** interface is the means of setting up the **DataStream** between the client and server making all **JSON** message communication possible. The **DataStream** is an implementation of a TCP connection, the **Socket** will initialise this stream and wait for a connection to be made to a client device, if no client connects before the timeout period then the connection will drop and needs reinitialising before a user can attempt to connect again.

4.4. Message types

Messages from server (events):

```
message template: {
        'type': string,
        'time': int,
        'id': int,
        'description': object
}
```

'type' is a unique string for each event type.
'time' is a the number of milliseconds that have passed since the beginning of the simulation in simulation time. Simulation time can consistent be real time but is usually not.
'id' is a unique id for this event. Included so that a client can tell that an otherwise identical event is not the same event.
'description' is a JSON object that contains required information about the event. The contents of the description object depend on the type of event. The formats of the description object for each event type are explained below.

type: "modelChanged"

```
description format: {
        "cars": [
                {
                "servicedFloors": [
                                int,
                ],
                "currentHeight": float,
                "id": int,
                "occupants": int,
                "capacity": int
                },
        ],
        "floors": [
                {
                "id": int,
                "height": float
                },
        ]
}
```

The 'modelChanged' event allows the client to initialise its model. Contains details about the cars and floors which will be referred to in later events by their id. This will always be the first event transmitted after the simulation starts. Clients should log the error and exit if this is not the case.

'cars': array of objects containing information about each car (elevator). Each object in the array has the following fields:

- 'servicedFloors': Array of ints. Each element is the id of a floor in the 'floors' array
- 'currentHeight': The current height of the car (unit unknown). This is always 0 at the beginning of the simulation
- 'id': A unique id for this car. Later events related to this car will refer to it by this id
- 'occupants': The number of people currently in the car. This is always 0 at the beginning of the simulation.
- 'capacity': The maximum number of people that can occupy this car.

'floors': array of objects containing the 'id' and 'height' of each floor. Floor heights use the same unit as car heights.

type: "carRequested"
description format: {
	'floor': int,
	'direction': '(up|down)'
}
Occurs when a person requests a car (not a specific car) at a floor.
'floor' is the floor at which a car was requested. 'direction' is the direction the person's destination is in. Only 'up' and 'down' are valid values.

type: "doorOpened"
description format: {
	'floor': int,
	'car': int
}
Occurs when a car's doors are opened. The car must be stopped at a floor for this to happen.
'floor' is the floor the car is stopped at.
'car' is the car whose doors have opened.

type: "doorClosed"
description format: {
	'floor': int,
	'car': int
}
Occurs when a car's doors close. The car must be stopped and its doors open for this to happen.

type: "doorSensorClear"
description format: {
	'floor': int,
	'car': int
}
Occurs when a door sensor is unobstructed after being obstructed. There is a door sensor at the door between each car and each floor. The car must be stopped at a floor with its doors open for this to happen.

type: "carArrived"
description format: {
    'floor': int,
    'car': int
}
Occurs when a car arrives at the floor it was last sent to.

type: "personEnteredCar"
description format: {
    'car': int
}
Occurs when a person enters a car. Upon receiving this event, a client should update its model to reflect the number of people in the car has increased by 1.

type: "personLeftCar"
description format: {
    'car': int
}
Occurs when a person leaves a car. Upon receiving this event, a client should update its model to reflect the number of people in the car has decreased by 1.

type: "floorRequested"
description format: {
    'car': int,
    'floor': int
}
Occurs when after a person has entered a car and enters their destination into the floor request panel. Clients should ensure the car eventually stops at the requested floor upon receiving this event.

type: "actionProcessed"
description format: {
'status': '(completed|inProgress|failed)',
'actionId': int,
'failureReason': string (optional)
}
Occurs when an action is processed.
'status': Whether the action has been performed, is being performed or will not be performed.
'actionId': The 'id' param that was sent when the client sent the action message.
'failureReason': Will be present if the status is 'failed'. Contains a human readable description of the cause of the error.

Messages from client (actions):

message template: {
'type': string,
'id': int,
'params': object
}
'type' is a string identifier for the type of action to be performed.
'id' is sent back to the client when success/failure should be reported. This should be unique for each action but can clash with event ids.
'params' is an object that contains information required to perform the action that is unique for each action type.

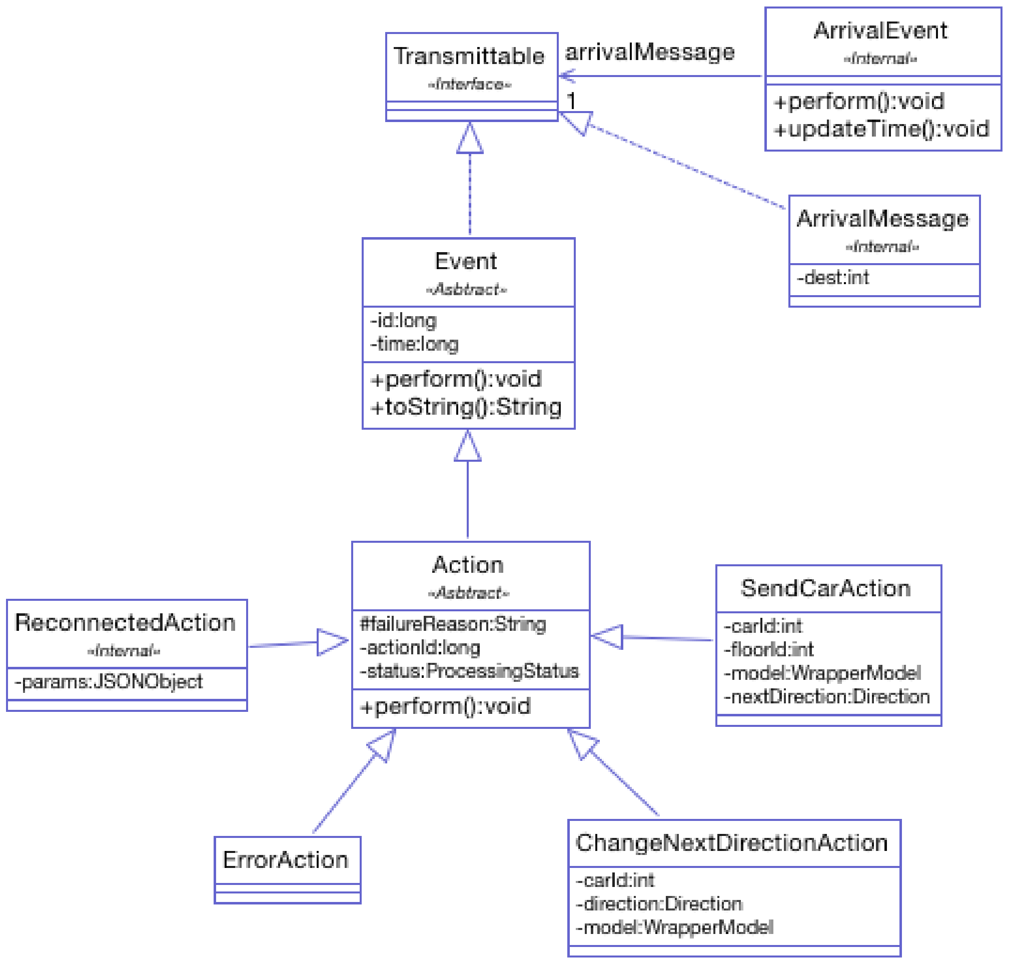
type: "sendCar"
params format: {
'car': int,
'floor': int,
'nextDirection': '(up|down)'
}
Sends car to the given floor. Car must be stopped and not have a destination set. 'nextDirection' determines which direction indicator will show when the car arrrives. This can be changed any time before arrival through the 'changeNextDirection' action. Success means that the car has left or is about to leave its current location. Clients should listen for 'carArrived' event before updating their model.

type: "changeNextDirection"

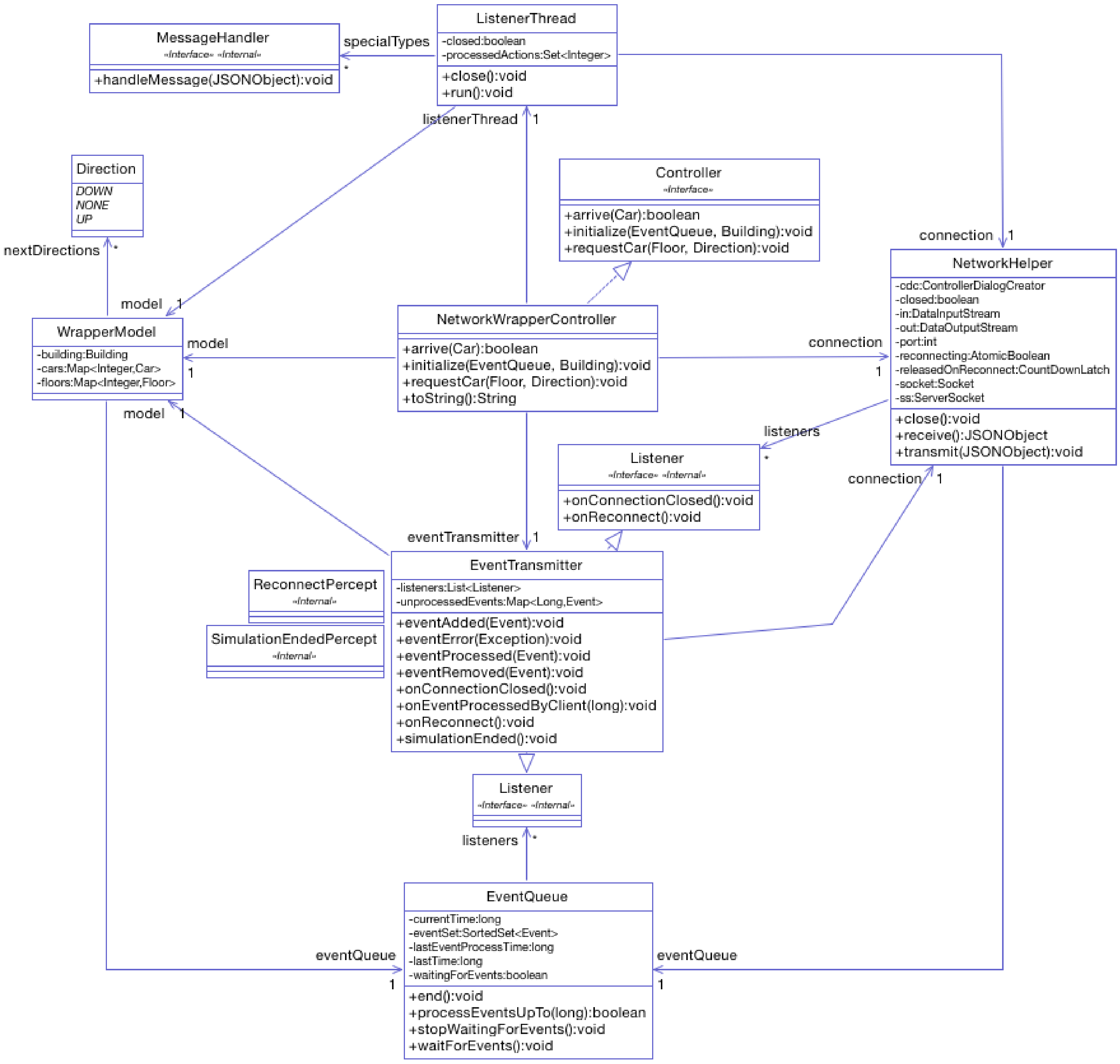
```
params format: {
	'car': int,
	'nextDirection': '(up|down)'
}
```

Changes the direction indicator that will show when the given car arrives at its destination. The car must have a destination set and must not have arrived yet. It should be noted that the direction indicator does not commit the car to travelling in the shown direction. It will, however affect whether people enter the car or not. Success means that the car has not yet arrived at its destination and when it does the given direction indicator will show.

Appendix: Class Diagrams



Perform - indicates that there is an event that is being performed, each class calls this anonymous method in the same way but the actions taken are different
Event - each event now has an **ID** that is used to discern it when being sent as a JSON message
ActionID - stores a unique identifier for each type of action within the classes
Status - stores the status of each event and returns whether each **Event** is going to be processed or not



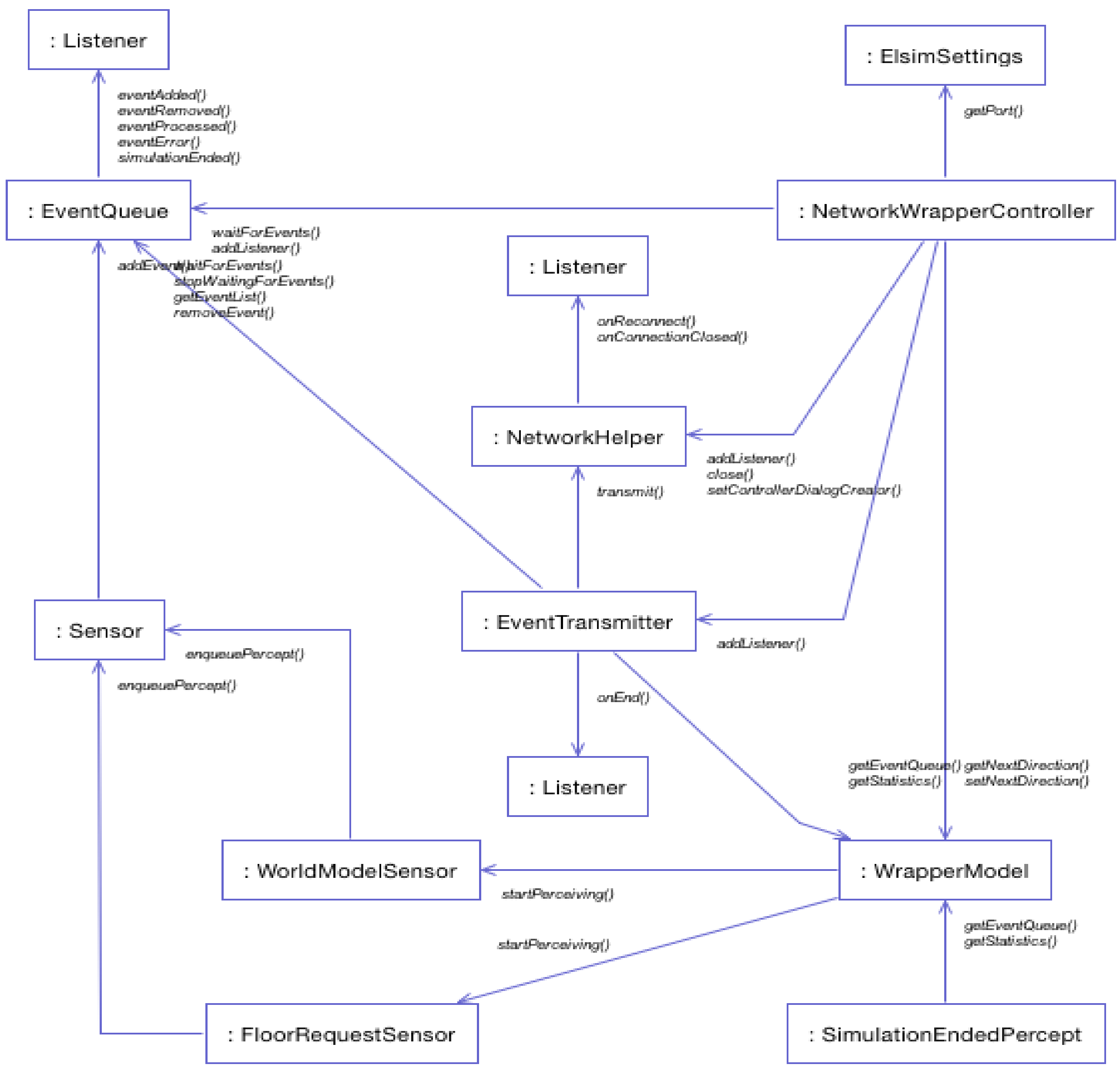
ListenerThread(ProcessedAction) - allows for the creation of JSON objects by sending the **type, description, time** and **ID** to the client across the **Socket**

NetworkHelper - handles the creation of the **socket** connection and creating of the network data transmission streams.

NetworkWrapperController(Initialize) - initially creates the building based on the user input parameters, also starts and populates the **Event Queue** with **CarRequests**

EventTransmitter - contains all the states that an event can be in the **Event Queue,** from when it is added to queue, whether it succeeded or failed to be executed and when the event is removed from the queue once it has been executed.

EventQueue - now has a means of detecting when events have stopped being transmitted with a timer in **waitForEvents** and once that timer has expired the **stopWaitingForEvents** the network connection will timeout.



This diagram represents the all the new and updated data structure of all Classes and interactions that we have developed over the length of the project