SARL ELEVATOR PROJECT: PART 1

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

Contents

1 *Background*

Elevator scheduling is a critical optimisation problem at the forefront of artificial intelligence and agent-oriented programming paradigms. The necessity of more efficient elevator planning systems is becoming increasingly apparent as modern, centralised and city-focused societies expand and become more complex.

Many simulators of real-world elevator systems, both open-source and proprietary, exist for the purpose of experimental testing of more effective planning systems. The simulator we have decided to extend for the purposes of integrating the **SARL Agent Programming Language** is openly available at http://elevatorsim.sourceforge.net/. This simulator will henceforth be referred to as the **Elevator Simulator.**

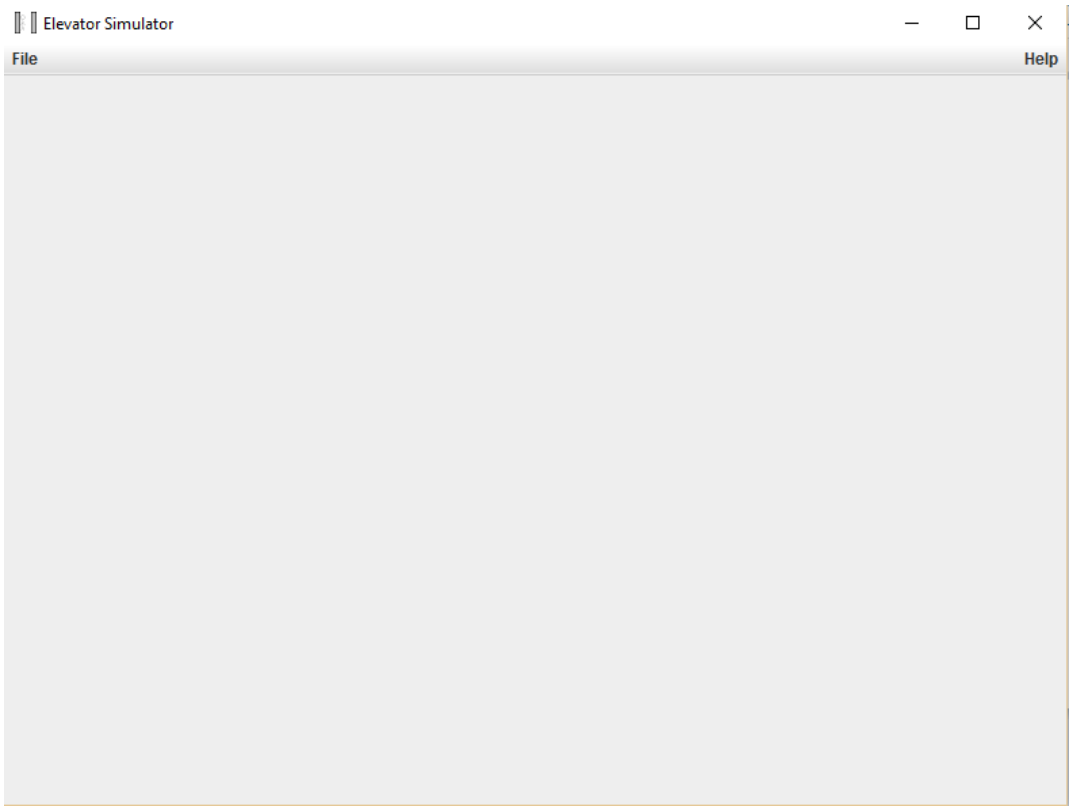
The Elevator Simulator is a Java-based simulation software which has remained in development since 2004. It allows for the testing of scheduling algorithms under a range of different predefined and user-customised scenarios, and provides access to a broad variety of simulated sensor data through an event system. The data exposed by the simulator includes **floor and car requests**, **elevator door open and close events** and even **tracking of the numbers of people who entered or exited the elevator**. Furthermore, the simulator crucially provides useful **statistics** after a simulation has been run.

2. Running the Simulation

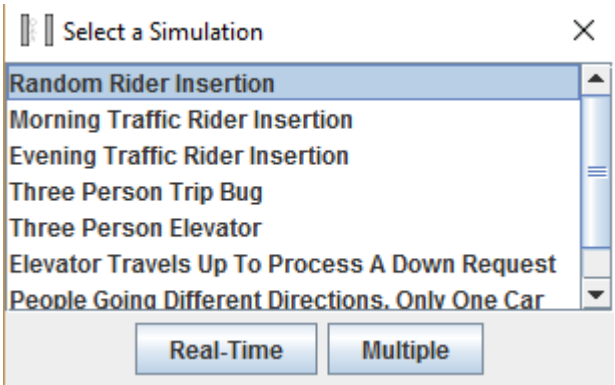
Following these steps will allow you to download and run the original build of the Elevator Simulator within an Eclipse workspace.

2.1. Download and running of the application

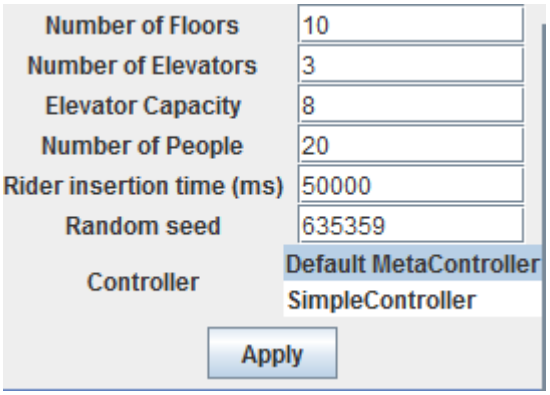
- You will need to download the original elevator simulator from https://sourceforge.net/projects/elevatorsim/files/elevatorsim/ElevatorSim%200.4/
- The file you will need is the elevatorsim-0.4.jar. If you wish to work with or look at the code you can download the elevatorsim-0.4-src.zip
- Once downloaded you can open the original simulator and will be presented with a blank window with a **File** and **Help** button, Click **File.**



- You will be presented with a range of scenario's (for details of each scenario refer to 3.1) for this this tutorial select **Random Rider Insertion** as the scenario and click the
- **Real Time** button. Note: other scenarios will lead to different output for subsequent steps

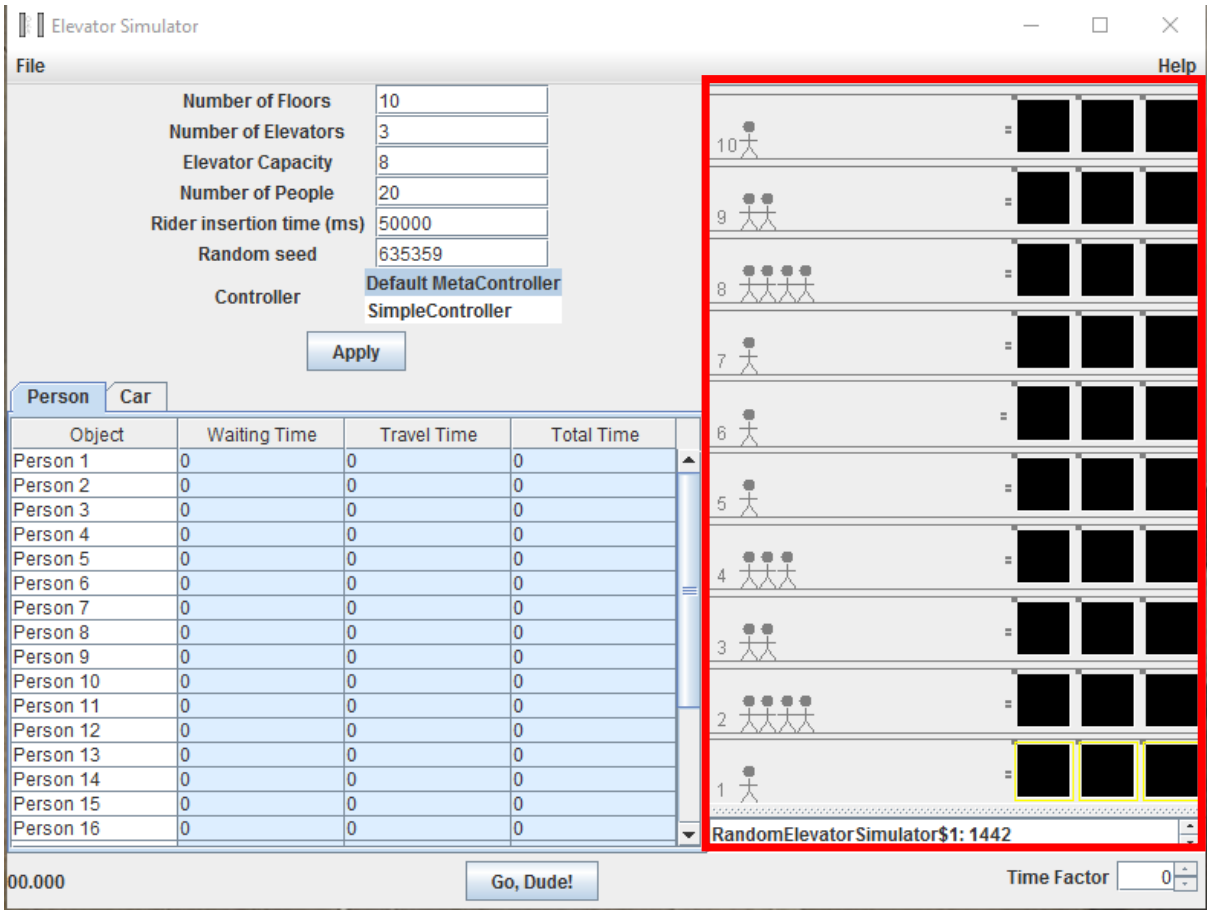


- The **Random Rider Insertion** will present the user with the following parameters



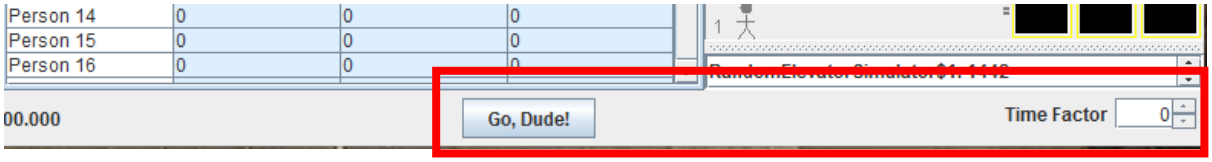
  - **Number of Floors** - How many Floors you would like to have in the building model
  - **Number of Elevators** - How many elevators will be servicing the floors
  - **Elevator Capacity** - The amount of people that can be in any elevator at a time
  - **Number of People** - The total amount of people randomly placed on floors making elevator requests within the building
  - **Rider Insertion Time** – Sets the time between each person making an Elevator request
  - **Random Seed** – Used to introduce randomness in the simulation
- The two controllers allow you to choose how you want the elevators to move throughout the building. (Refer to 3.3.)
  - The **Simple Controller** sends the elevator to every floor in the direction that it is moving to check to see if there is a request from a person heading the same direction
  - The **Meta Controller** runs an algorithm that tries to decide which elevator is best suited to service each floor.

- Click **Apply** and the parameters will be applied and represented

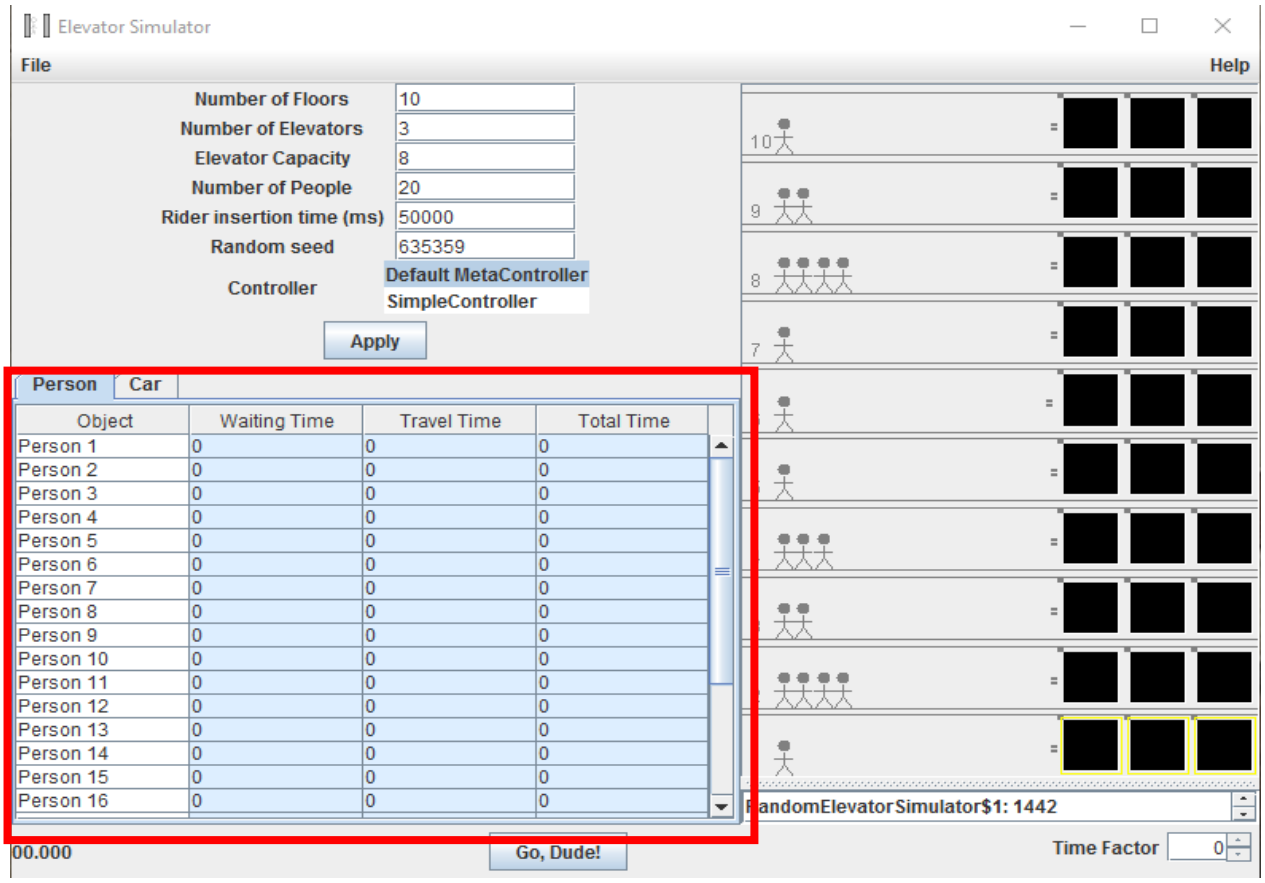


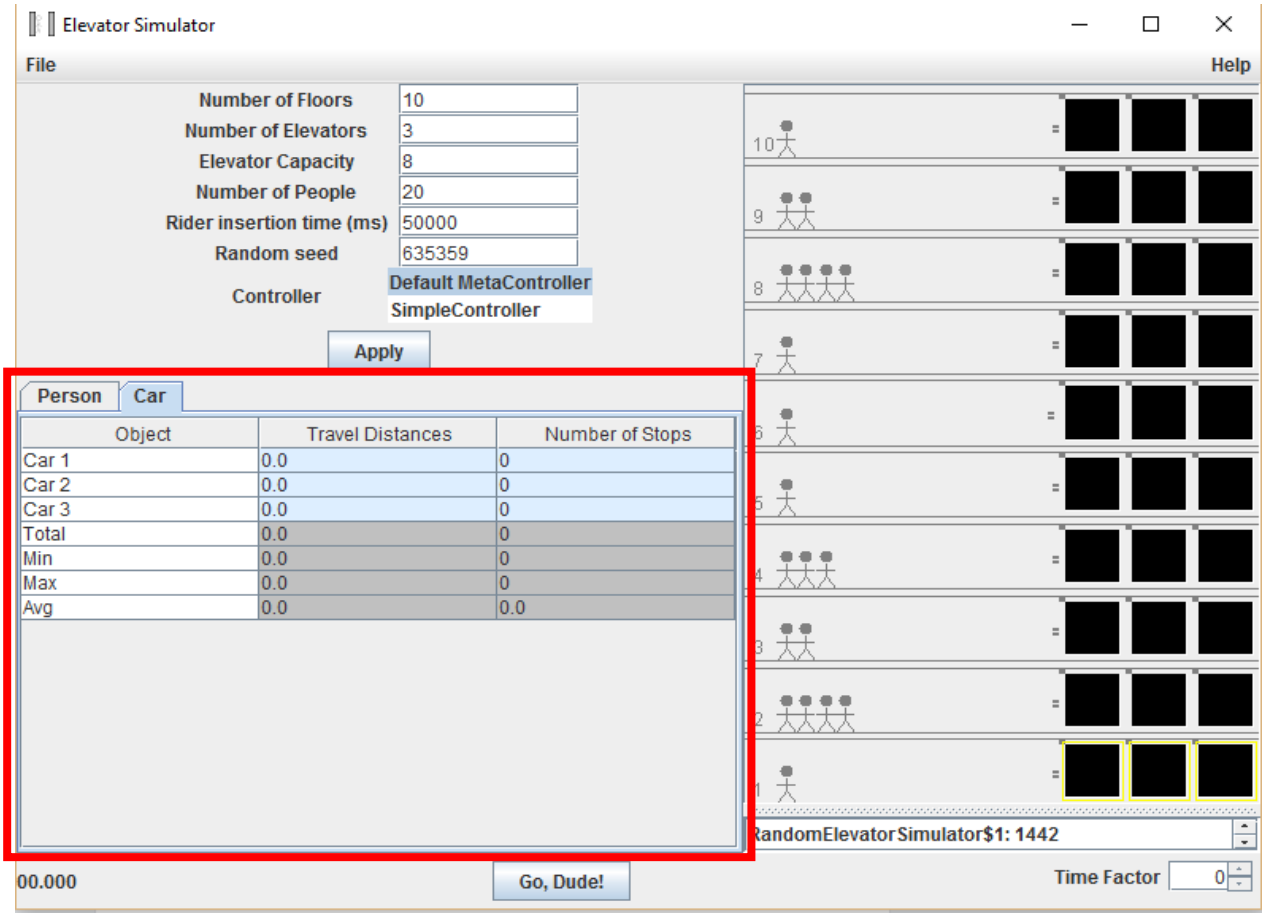
- On the right you will see the Elevators and people you entered previously
- Underneath the elevators and people you will see the **Events** that have been passed through the **Event Queue** (refer to 3.4.) which you can cycle through

- The next step is to click **Go,Dude!** The simulation will start in the normal speed, to increase the running speed of the you can toggle the **Time Factor** arrows to the right



- The simulation will run until all Car and Floor Requests, throughout the simulation you will notice the **People** are **Car** tab will be changing. This represents different data and statistics which are being collected about each Person and Car





3 *The Simulation*

3.1. Initialising Simulation and Scenario Selection

Package: **org.intranet.elevator**

The simulation begins by creating the building set up in the simulation. From there the user is shown the set of simulations that have been predefined by the original development team. A Simulation simply determines when People will call elevators and from what floor to what destinations. The Scenario that is selected also dictates what parameters of the simulation will be user configurable (such as number of floors, number of cars, etc.).These simulations represent a variety of every-day operations and situations, such as a morning rush of people needing to move from the bottom floor to their respective floors or people on many floors needing to navigate to other floors within the building.
Once a simulation is selected and the parameters have been set by the user the **event queue** is populated with every **carRequest** from each person that the simulation has generated

The default Simulators that are available are:

- **Random Rider Insertion**: This allows the user to set the number of people, elevators, floors and capacity of each elevator. The people calling the elevator are placed on random floors, it also allows the user to set up the delay of when a random person hits the elevator call button. This is also the only simulation that allows the user to choose which controller they want to use.
- **Morning Traffic Elevator**: allows the user to set the number of floors, elevators, people, and the delay between each set of people arriving at the elevators. All people will spawn at the bottom floor and the meta controller (refer to 3.3.2) is used in this scenario
- **Evening Traffic Elevator**: This simulation allows the user to set the same parameters as the Morning Traffic scenario, but the people are all destined for the ground floor to exit the building
- **Three Person Bug**: This scenario spawns only 2 people on different floors in which only one person has called the elevator, then allows the user to set the time in which the second person calls the elevator
- **Three Person Elevator**: This simulation allows the user to set only the number of floors and elevators, it spawns only three people on different floors with no delays
- **People going different directions**: The user here is asked to set the number of floors and elevators, there are only 2 people on separate floors and the user has to set which floor each of the riders is going to. This scenario allowed for testing of the meta controller and its execution of handling priority of movement
- **Three People Two Elevators**: This allows the user to set the floors and number of elevators, this scenario deals with two people heading in one direction and one heading in the other, we feel that this was also used for testing and updating of the meta controller

3.2. Building, People and Model Construction

At the beginning of each simulation, the building, people and floors are generated, fulfilling the parameters set by either the user or Simulator previously mentioned.
The **EventQueue** is populated with all the **CarRequestEvents** (these requests are a person on a floor that is calling a **Car** to come pick them up, this call also has a **Direction** that is passed to each **Car** so that the **Car** knows how many people are going to be entering the **Car)** required for the entirety of the simulation.

3.3. Controllers

Package: **org.intranet.elevator.model.operate.controllers**

A **Controller** is an interface that defines several methods meant to react to certain occurrences in the simulation. For example, **carRequest,** an **event** inside the controller, is called when a Person calls an elevator to a floor. It is meant to determine which car to send and whether to send it immediately or later. Controllers define when
and where to send cars. It was created so that different elevator control algorithms may be defined and tested easily.
The two basic Controllers provided with the Elevator Simulator are **SimpleController** and **MetaController**.

3.3.1. The Simple Controller

Source: **SimpleController.java**
Package: **org.intranet.elevator.model.operate.controllers**
The **simple controller** has a check that runs on each floor that looks to see if there is a person that needs to be picked up or if they need to be dropped off based on the floor that they are on and the direction that the car is moving.
The simple controller has no way to prioritise which elevator is to be sent to pick up any particular passenger, if there is space and the elevator is heading in the right direction a person will be picked up.

3.3.2. The Meta Controller

Source: **MetaController.java**
Package: **org.intranet.elevator.model.operate.controllers**
MetaController is an attempt at a more intelligent elevator control algorithm. When a car is requested, it determines the best car to send based on their locations and the directions are travelling.
It uses a class called **CarController** to keep track of a queue of destinations the car it is tied to must stop at. Destinations are added when **MetaController** determines it is the best car to satisfy a car request or when a Person enters the car and requests a floor. The car will travel in one direction and only turn around once there are no more destinations in that direction.

```
for (CarController controller : carControllers) {
       float cost = controller.getCost(floor, direction);
       if (cost < lowestCost) {
                           c = controller;
                           lowestCost = cost;
                     } else if (cost == lowestCost) {
                           if (controller.getCar().getTotalDistance() <
       c.getCar().getTotalDistance())
                                                 c = controller;
```

3.4. Simulator Events

3.4.1. Event Specification

Here is a list of events that can be pass between the classes and objects that are involved with receiving and transporting people between floors within the building

Person
Source: **Person.java**
Package: **org.intranet.elevator.model.operate**

- **Person.setDestination() Event Wrapper -** sets the destination to one of the other floors in the simulation
- **Person.LeavingCarEvent -** indicates that one person has left an elevator
- **Person.EnteringCarEvent -** indicates that one person has entered an elevaotr

Door
Source: **Door.java**
Package: **org.intranet.elevator.model**

- **Door.OpenEvent -** After **carRequest** is performed to get to target floor the door will open
- **Door.CloseEvent -** Once all **carRequests** have been filled the door is cleared to close

DoorSensor
Source: **Door.java**
Package: **org.intranet.elevator.model**

- **DoorSensor.ClearEvent -** Sets `DoorSensor` to unobstructed by people and door available to be closed

MovableLocation
Source: **MovableLocation.java**
Package: **org.intranet.elevator.model**

- **MovableLocation.ArrivalEvent -** Sets time at which a movable location should arrive at its target destination

3.4.2. The Event Queue

Source: **EventQueue.java**
Package: **org.intranet.sim.event**

The **EventQueue** is central to the operation of the application. It is a singleton that every model class has access to. When any model class wants something to happen at a certain time (in simulation time), it creates a new **Event** and adds it to the **EventQueue**. An Event specifies a time it should be performed and defines a **perform** method.

The **Perform** is an anonymous method that is called upon when an event needs to be processed, it has no set functionality but is more an indication that an event is being performed by the system. The **Perform** method is called in many source files within the project and performs the events such as opening the **Door** of a **Car,** setting the destination floor for a **Car,** declaring that **Person** has entered or left a **Car** and many more

The EventQueue will ensure perform is called at the specified time. A key example of this is a Simulator adding several instances of **CarRequestEvent** in its **initiliazeModel** method. This event simply causes a person to request a car.

3.5. Navigating the Building

3.5.1. Entering the car

Source: **/build/src/app/org/intranet/elevator/model/operate**

At each floor there are a set of events that are passed through the event queue so that the elevator knows whether it is ok for a particular car to move to the next floor.

1. Once arriving at the next **DestinationFloor** the **Door.OpenEvent** is **performed** by the **EventQueue,** using the appropriate implementations of **perform,** so that people can enter and exit the car.
2. The doorway is then checked for obstructions. If there are more people going in than the elevator's capacity, then the door is obstructed and the door will not receive **Door.CloseEvent**
3. Then, the door becomes closed, clearing away the unobstructed event, meaning that no one can move in or out of the elevator while it is moving.

3.5.2. Setting a new Destination

Source: **/build/src/app/org/intranet/elevator/model/operate**

1. Once all the people have entered the car the people who have reached their **DestinationFloor** will perform a **LeavingCarEvent** and the car will receive a **SetNextDestination(Floor) Event**
2. Then, set the next destination based on the closest floor to its current location in the direction that it is travelling.

3.6 Timer and Statistical Data

3.6.1. Setting and updating time

Source: /build/src/app/org/intranet/sim/clock/Clock.java

Each event in the simulation is timestamped when it is triggered from the event queue using the **updateTime** method, this method returns the time since the simulation has started. This data is used to return statistical data for the time that people have had to wait for the lift, the time they have spent on the lift and total time.

3.6.2. Person statistics

Source: /build/src/app/org/intranet/elevator/model/operate/Person.java

- **Person.beginTravel -** gets current simulation time for each person that has entered a car
- **Person.endTravel -** gets the current simulation time and minuses **beginTravel** for each person leaving a car
- **Person.startWaiting -** gets the current time once a **Person** has sent a **carRequest event**
- **Person.endWaiting -** the time between when a **Person** sends a **carRequest** and the when car arrives

Each person that has used an elevator within the simulation has these methods returned as raw time and the total of all these areas, a minimum, maximum and average for all these times is also generated.

3.6.3. Car statistics

Source: /build/src/app/org/intranet/elevator/model/Car.java

- **MovableLocation.numTravels** - stores the amount of **floor** objects the car picks up a **Person** from
- **MovableLocation.totalDistance -** stores the total distance in **Floors** that each **Car** has travelled

Each car also has its own data stored for each run of the simulation, the amount of stops that each car has made and the total travel distance is stored and like the people data, the total, minimum, maximum and average of these areas is returned.
All of this data is represented in milliseconds.

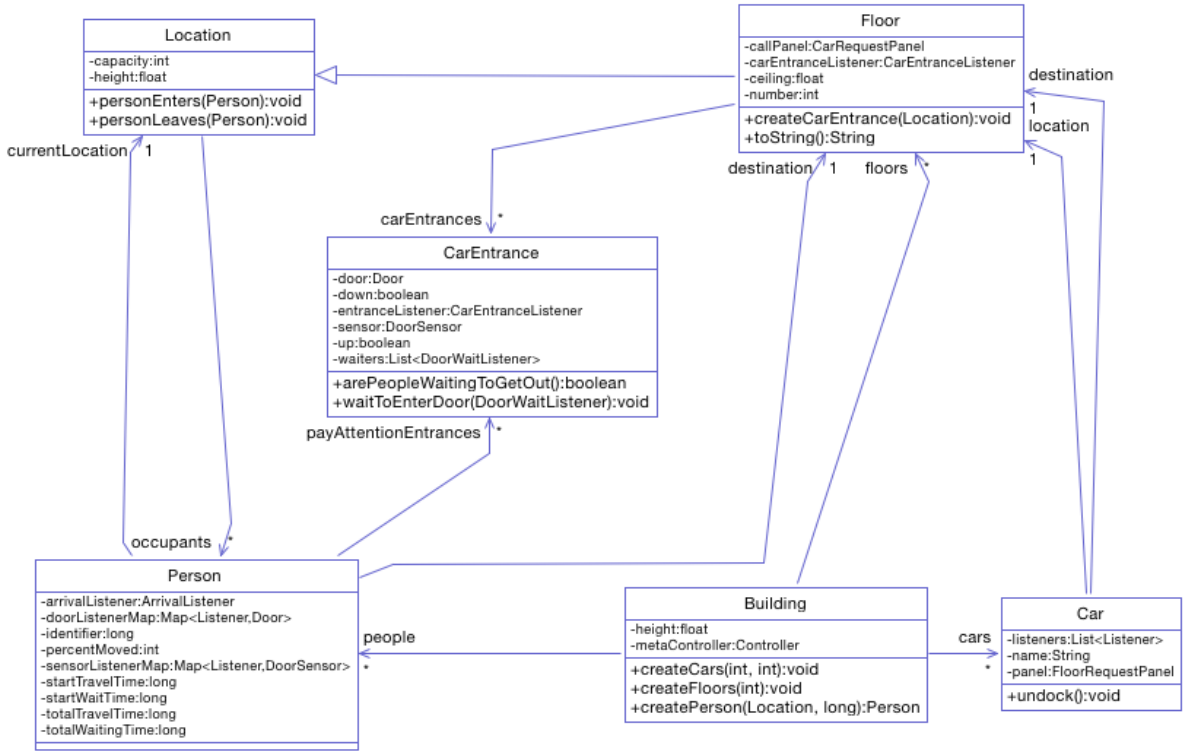
3.6.4. Timer

Source: /build/src/app/org/intranet/sim/clock/RealTimeClock.java

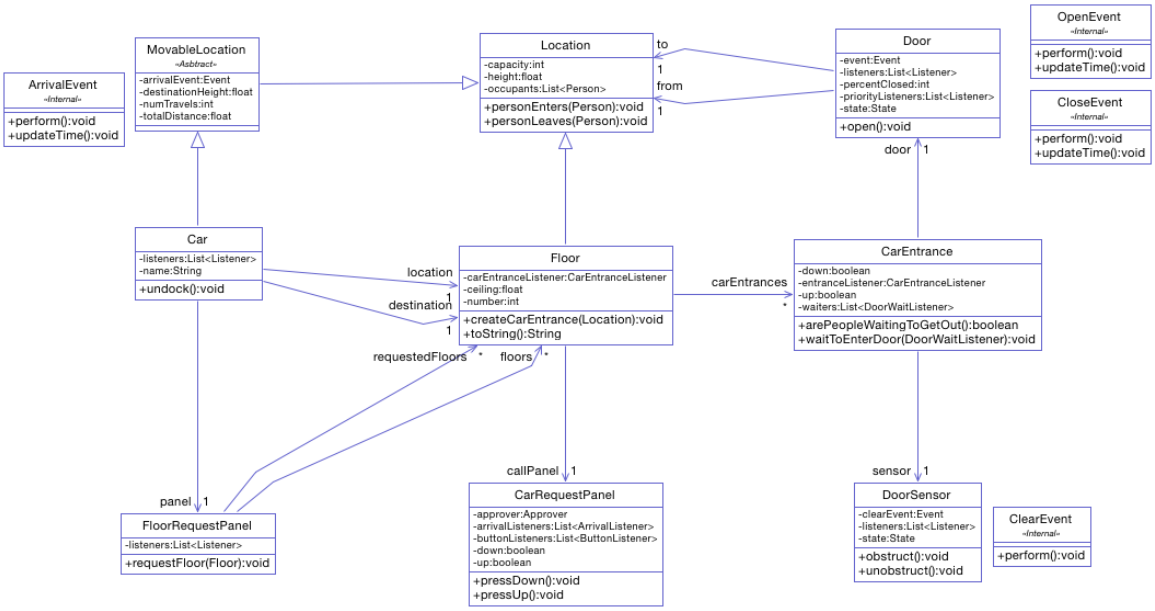
Each simulation is run on a timer, this timer is started at the start of each simulation. As discussed is used for the statistical data when called in the **Person** and **MovableLocation** class with the previously discussed methods.
The timer can be modified to speed up and slow down. This timer follows a simple 0.1 milliseconds to **Math.pow(2, accelFactor),** where accelfactor is an integer that is scalable throughout the simulation.

Appendix: Class Diagrams

overallClass

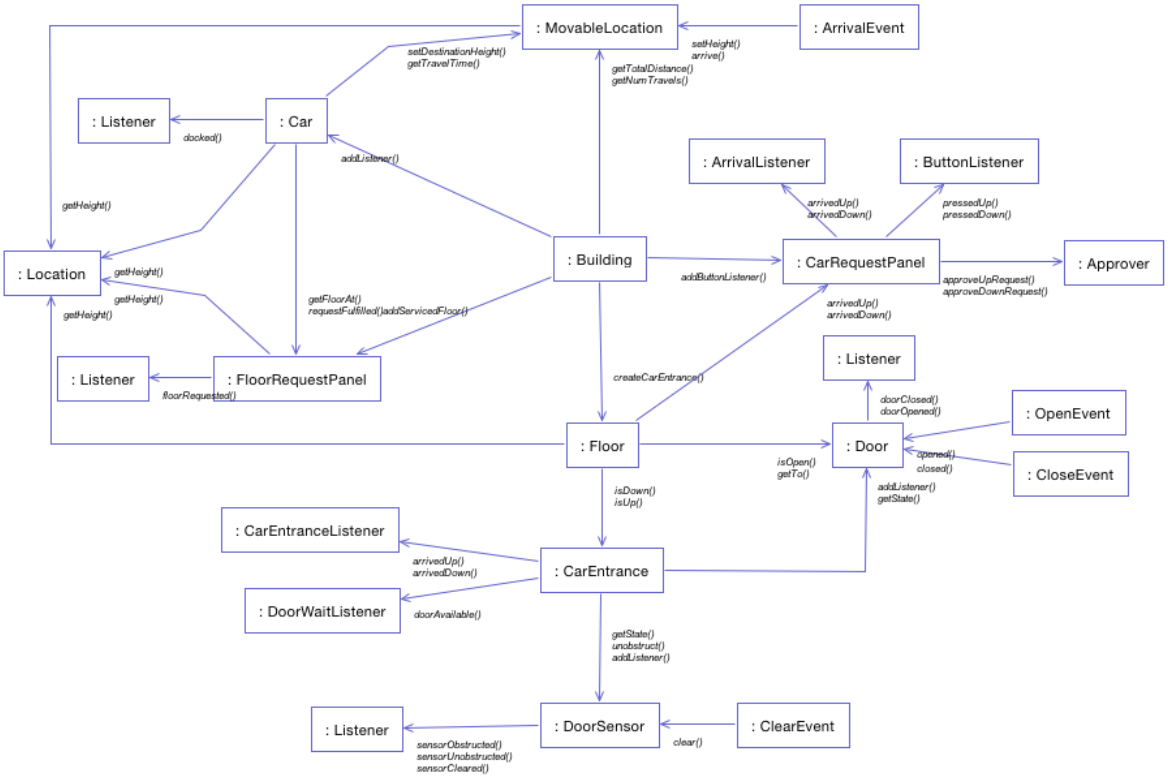


Model Class



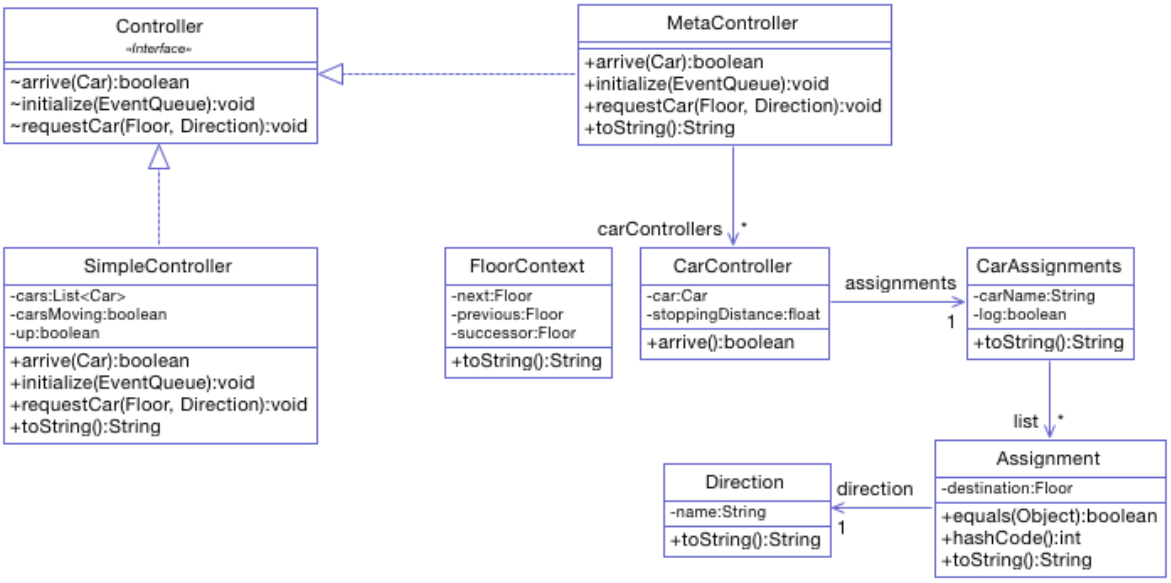
The Model Class holds information about:
Car – This Class holds information about each elevator **Car** including the **CarID**
CarRequestPanel – Allows the **Person** to send a **FloorRequest** to move to within the simulation
MovableLocation – This allows the **Car** to move between the **Floors** to fulfil requests
Location – this stores the information about the building, including **Floors** and **People**

Model interaction



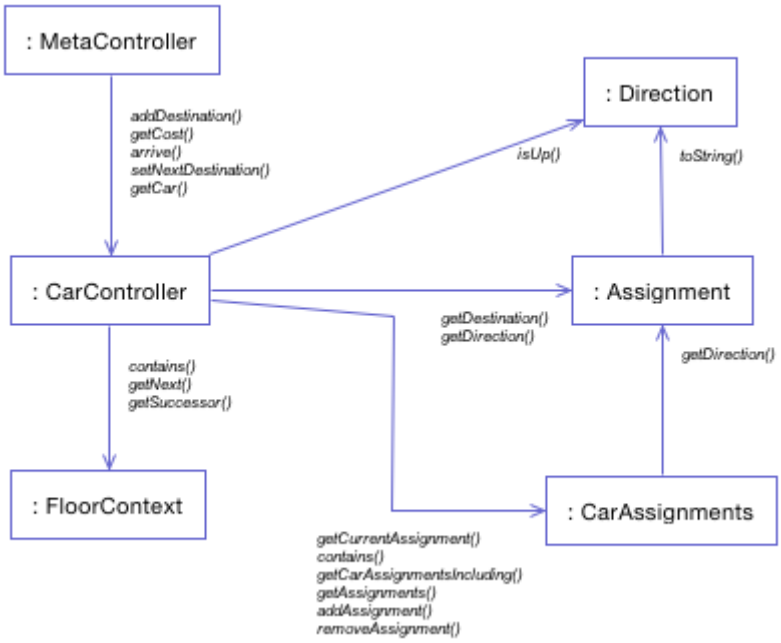
Represents all interactions and data structure of the Model with its extended and related classes

Controller Class



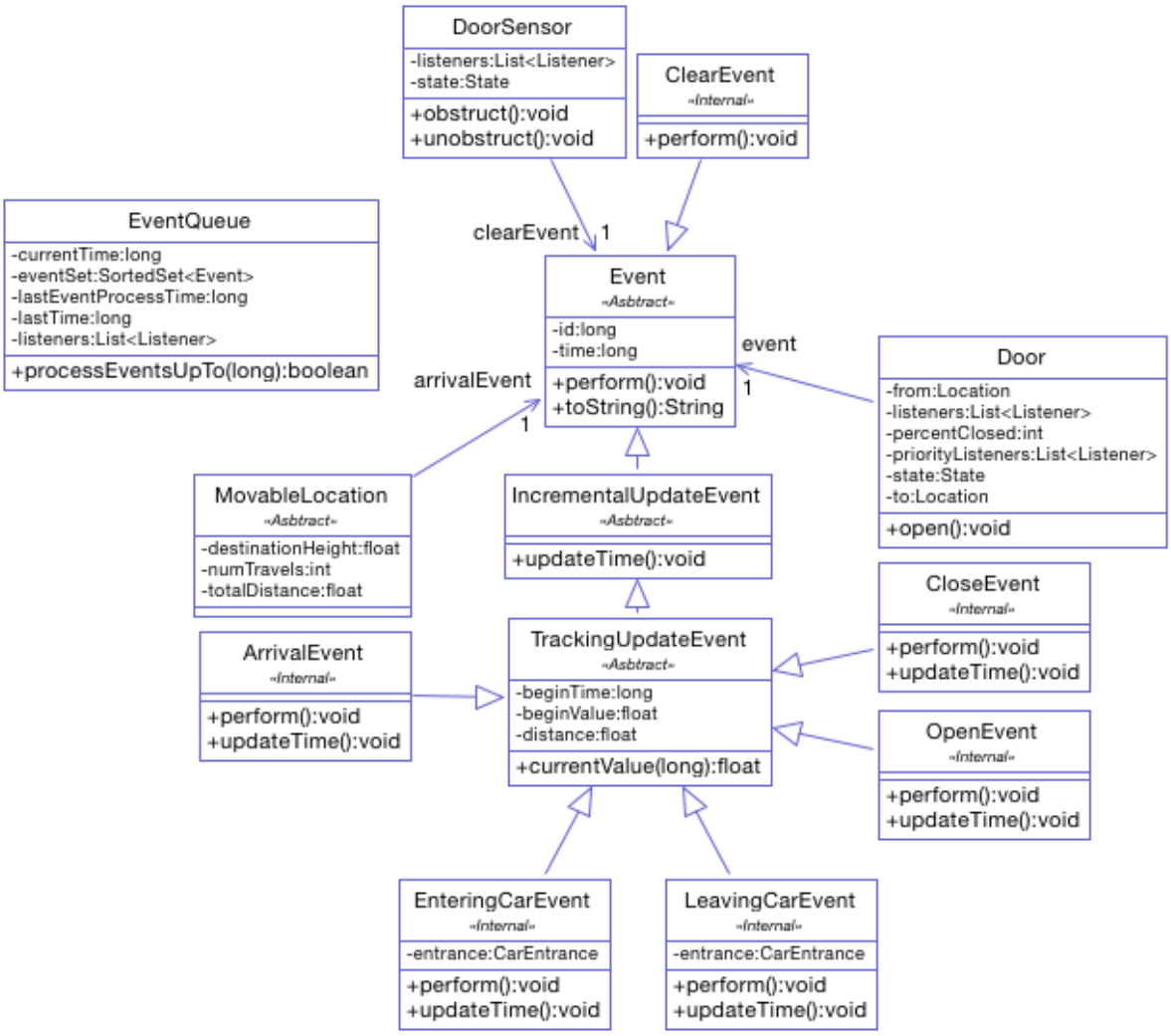
Controller – calls the car through **Requests** and **Arrival** events at each floor, also initialises the **Event Queue.**
Simple Controller – access the **controller** class and passes its methods to allow the **event Queue** to be processed
Mete Controller – Accesses **Event Queue** as well but allows other methods and implements **Assignment** to allow the simulation to find the best car for each request

Controller interaction



Represents the actions that the Controller have with each other and shows the methods that are used across the multiple classes

Event Class

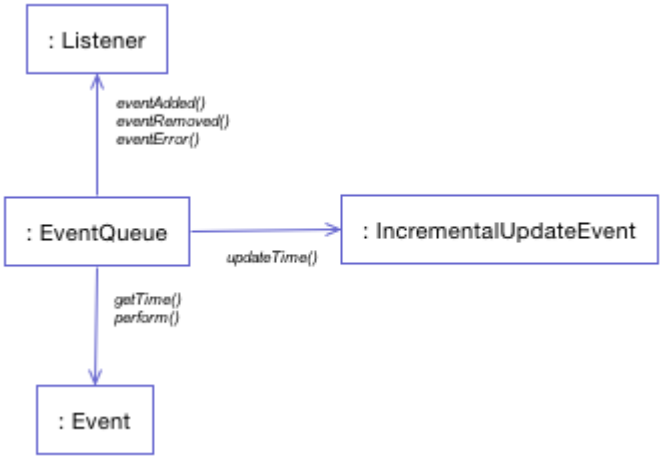


Event Queue – allows each **Event** to be **Performed** in the correct order to complete the simulation, also allows the simulation to know when to stop running any processes when the **Event Queue** is empty

Tracking Update – Ensures that the classes that are to be passed into the **Event Queue** are in the functional order to mitigate errors

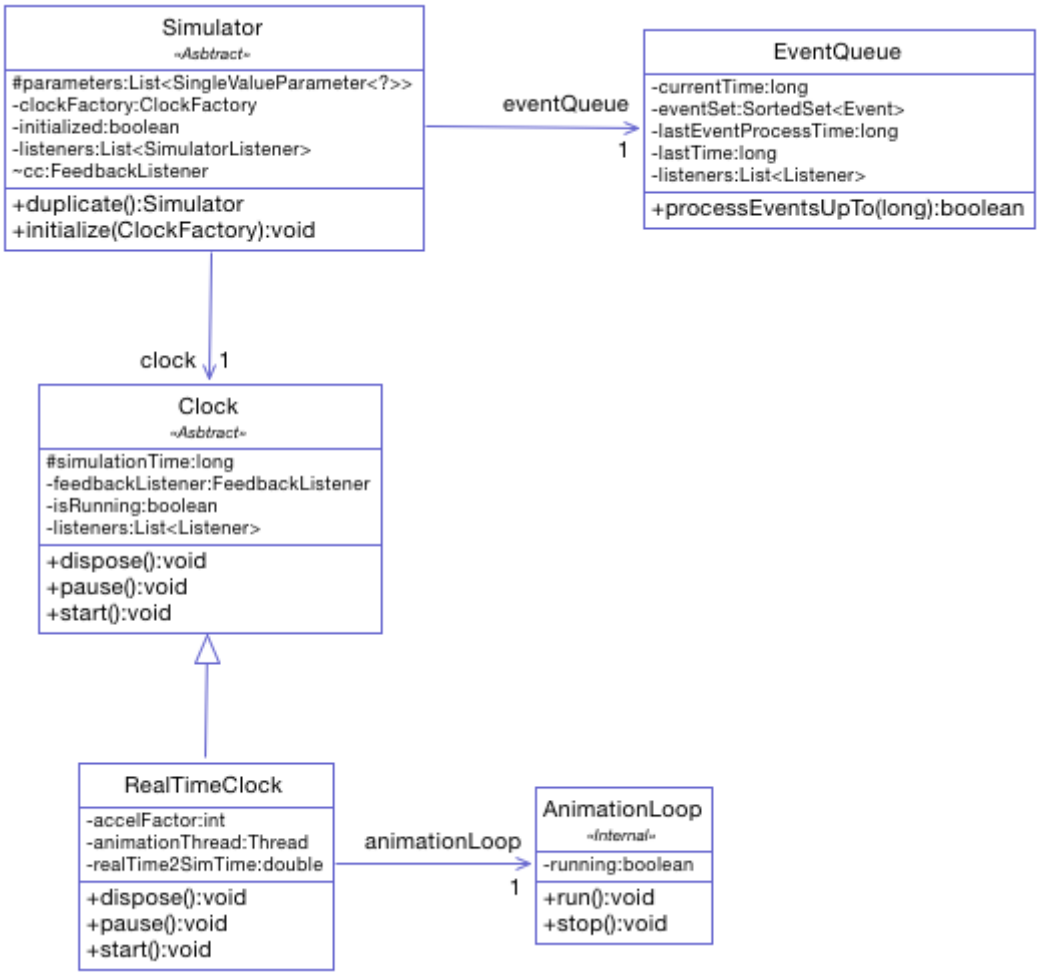
Event – takes every action from other classes to be **Performed** and performs then, also stores the time log of when each event occurs

Event interactions



Represents how the Events are passed through the **Event Queue** and how the system updates itself with timestamping **Events** that are performed

Clock Class



Clock – Holds the current run time of the simulation and allows the **Event Queue** to timestamp what time each event in simulation time is **Performed**
RealTimeClock – allows the Speed of the simulation to be accelerated by a factor of the current time

Clock interaction