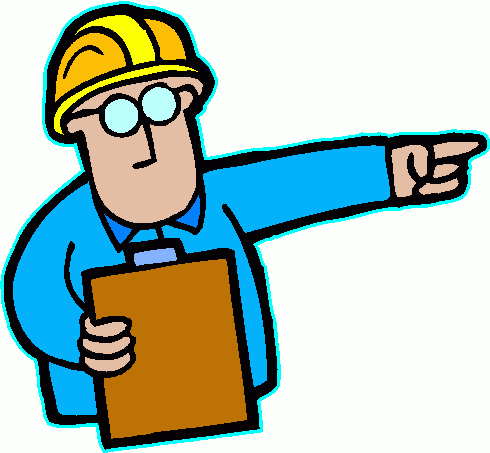
# Parking Lots Simulation

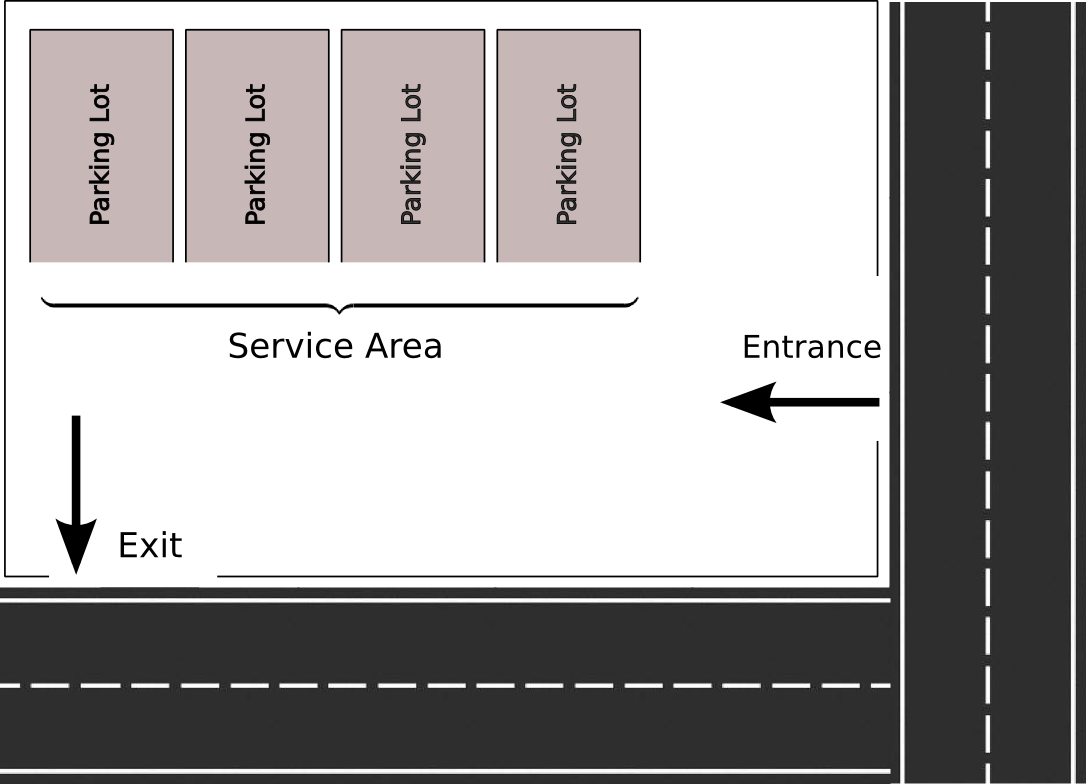
Sami is a supervisor for a big inventory and he manages the traffic for vehicles coming in, he has received many complaints and noticed many delays and inefficient usage for the inventory parking lots and he decided to find what causes such delays to identify the problems then solves them.



He keeps records for each day for the incoming vehicles and he wants your help to implement a simple simulation (as a starting point) through which he can enter those records and run a simulation for serving those records.

Since he has some programming experience, he implemented a simple UI (named case generator) in which he defines inputs (configuration) and incoming vehicles arrival time and tasks to be done, he needs you to take those inputs then run a simulation to serve the incoming vehicles according to his inputs (parameters).

He has two types of vehicles: **trucks and vans**, and each vehicle coming to the inventory would do **loading/off-loading** for the items from/to the inventory, he has specific numbers of parking lots inside his inventory where the vehicles can park and serviced by the inventory workers (who would do loading and off-loading of items), the vehicles coming to his inventory would be serviced only if they are parked at one of those parking lots, if a vehicle comes in and finds no empty parking lots it would wait outside the inventory until one of the lots is empty.



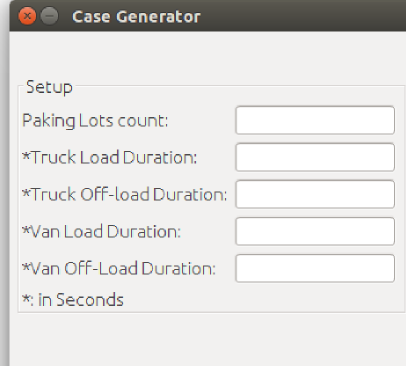
Depending on his experience, he has the average time required to complete a task for each vehicle type and he controls those values through the UI

A vehicle coming to the inventory would go into the following steps:

* Waiting: if the parking lots are full the coming vehicle needs to wait until one of them becomes empty, vehicles wait in a queue according to their arrival time and they will be served accordingly.
* Parked: When a waiting vehicle turn is due and there is an empty parking lot, it should move to the empty lot to be serviced (load or offload) by the workers, at this stage the vehicle state is parked.

The vehicle shall remain in that step according to duration configured through the UI, for example: if Sami sets the time for a truck doing load for items needs 90 seconds, it shall occupies that single parking lot for the whole 90 seconds.

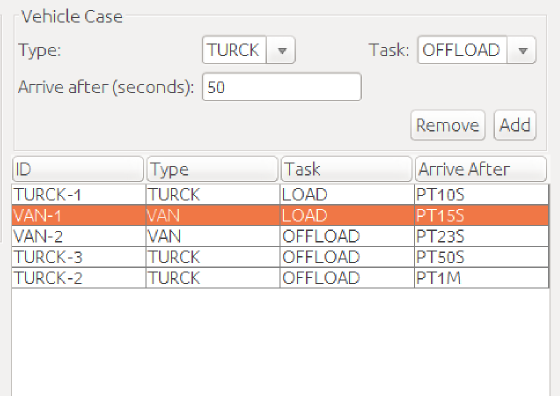
* Left: when the vehicle (loading or off-loading) task is completed, it leaves the inventory to allow other waiting vehicles to be served.



Sami defines in the generator the inputs for the simulation by defining the following:

* Parking lots count: defines the number of parking lots available, currently he has only four but he needs to see the effect if he decided to add or remove lots. Vehicles are served in parallel, for example: if the number of lots is set to five and it is rush time and you have many vehicles in the waiting queue, you shall always have five vehicles at service area occupying the five lots.
* Tasks duration fields: in those fields he defines the time needed (in seconds) for each vehicle type to complete a specific task:
  + Truck Load Duration: defines the amount of time needed for a truck to load items
  + Truck Off-load duration: defines the amount of time needed for a truck to do off-loading for items.
  + Van load duration: defines the amount of time needed for a van to load items
  + Van off-load duration: defines the amount of time needed for a van do off-load items.

In another section of the generator UI he defines the vehicle cases (records) to be simulated by defining the vehicle type, the task type, and its arrival time, instead of specifying at what time did a vehicle arrived he chosen to insert at which second of the simulation shall the vehicle reach to the inventory.



After completing the inputs he clicks the “Start” button to run the simulation, a dialog is shown to display and monitor the progress of the simulation and its status.

He defined the contract that you need to implement for the simulation, in his code it is invoked after the case generation and passes it a reference of type ***SimulationCase*** which holds the case configuration:

***Simulation*** interface defines the simulation strategy which should run the simulation, it shall accept instances of type ***SimulationListener*** interface that should by called by the simulation when it’s state changes:

/\*\*

\* Defines a simulation execution strategy. The assumption is to create a new

\* instance for each simulation run.

\*

\* **@author** Sami

\*

\*/

**public** **interface** Simulation {

/\*\*

\* Pass the simulation the case to test, <code>simCase</code> contains all

\* information needed to run the simulation, the tasks duration for each

\* vehicle type and task and the arrival time.

\*

\* **@param** simCase

\* the case to test and run

\*/

**public** **void** start(SimulationCase simCase);

/\*\*

\* Gets a snapshot for the simulation execution started through

\* {@link #start(SimulationCase)} and where it is standing, returned object

\* shall tells how many parking lots are occupied and the arrival time for the

\* vehicles and the status for each one in the simulation.

\*

\* **@return** a snapshot from the current execution status.

\*/

**public** SimulationState getCurrentState();

/\*\*

\* Add a listener to this simulation to get notification for any update

\* happening inside the simulation, listener methods receive instances of

\* {@link SimulationState}

\*

\* **@param** listener a listener to simulation execution

\*/

**public** **void** addStateListener(SimulationStateListener listener);

}

***SimulationState***, produced by your ***Simulation*** interface implementation which shall give a representation for your current simulation state:

**import** java.time.LocalDate;

**import** java.time.LocalDateTime;

**import** java.util.stream.Stream;

**import** java.util.stream.StreamSupport;

/\*\*

\* Defines a representation for a simulation status.

\*

\* **@author** Sami

\*

\*/

**public** **interface** SimulationState {

/\*\*

\*

\* **@return** the related simulation case

\*/

SimulationCase getCase();

/\*\*

\* Number of parking lots currently occupied

\*

\* **@return**

\*/

**int** occupiedParkinglots();

/\*\*

\* The start time of the simulation.

\*

\* **@return** the start time of the simulation

\*/

LocalDateTime startTime();

/\*\*

\* Complete time of the simulation.

\*

\* **@return** simulation completion time or <code>null</code> if it is still

\* running.

\*/

LocalDateTime completeTime();

/\*\*

\* Returns if the simulation is still running, a simulation is considered

\* running if it stills has vehicle cases not served.

\*

\* **@return** <code>true</code> if the simulation is still running.

\*/

**boolean** isRunning();

/\*\*

\* Returns a representation for vehicle statuses.

\*

\* **@return**

\*/

Iterable<VehicleStatus> getVehicleStatuses();

**default** Stream<VehicleStatus> getVehicleStatusesAsStream() {

Iterable<VehicleStatus> iterable = getVehicleStatuses();

**return** StreamSupport.*stream*(iterable.spliterator(), **false**);

}

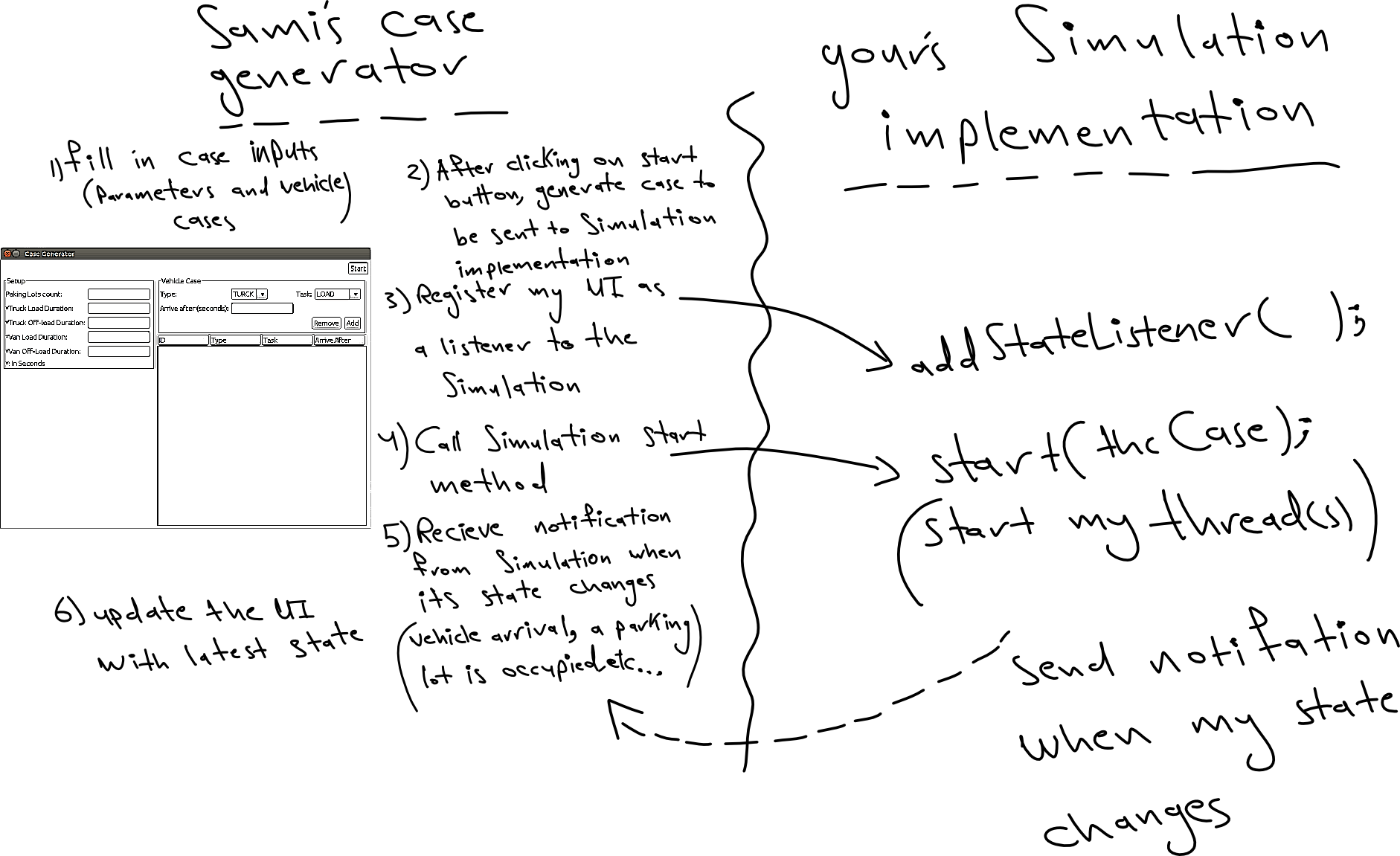
**default** **boolean** hasEmptyLots() {

**return** getCase().parkingLotsCount() != occupiedParkinglots();

}

}

Those interfaces were introduced by Sami and they are used through his code, below image describes how this is done:



The case generator implemented by Sami generates an instance of ***SimulationCase*** that holds the simulation inputs and vehicle cases then sends it to the Simulation implementation which should then starts an internal thread(s) to simulate the case, the simulation shall sends notification to registered listener(s) when any change in the simulation occurs.

The notifications received from the Simulation are instances of ***SimulationState*** which describes the current state of the simulation (number of occupied lots, the status of vehicles, if the simulation is completed or not, etc...).

There is one more thing Sami’s is asking for, he wrote the case generator in a rush so his code is not clean and he asking your help to refactor his code to look better.

***You will be provided with all classes and interfaces written by Sami along with this document.***