DEPARTMENT OF COMPUTER SCIENCE & ENGINEERING
UNIVERSITY OF NEBRASKA—LINCOLN

NetFlow

Traffic Simulator

Zoe Fu, Reid Stagemeyer, and Dat Nguyen
03/01/19
Version 1.0

This document provides details of the new NetFlow traffic simulator system for Pixi City

Revision History

Version	Description of Change(s)	Author(s)	Date
1.0	Initial draft of this design document	Nguyen, Fu, and Stagemeyer	2019/01/03
2.0	API update	Fu, Nguyen and Stagemeyer	2019/08/03
3.0	Detailed Design Update	Fu, Nguyen and Stagemeyer	2019/25/03

Contents

1. Introduction

The NetFlow Traffic Simulator software system provides functionality that will allow Mayor Mann to more efficiently conduct city planning/design. Due to ever tightening budget restraints and the increased emphasis on logistics it was deemed necessary to develop a simulator that will allow the city to develop more efficient designs.

The new IMS system is a Java-based Object-Oriented design that allows for more efficient data handling for the client. This new system will provide a new way to monitor and manage invoices that will replace the old legacy system.

1.1 Purpose of this Document

This document aims to describe the detailed system design, architecture, and interface of the Traffic Simulator. Detailed descriptions are given including those of individual components and subsystems.

1.2 Design goals

The overall design goal for this project was to allow Mayor Mann to run multiple simulations of traffic flows based on different combinations of traffic components (stop lights, stop signs) throughout the city. This will allow him to find the optimal placement of said components in order to get cars to their destinations as quickly as possible.

1.1 Design trade-offs

It was decided to allow the maps to be created dynamically via a .csv file that will allow for less software overhead and more consistency. Database connectivity wasn't chosen to minimize connectivity issues and redundancy but could limit future scaling.

1.2 Interface documentation guidelines

- Classes are named with singular nouns.
- Methods are named with verb phrases, fields, and parameters with noun phrases.
- Error status is returned via an exception, not a return value.

1.3 Definitions, acronyms, and abbreviations

OOP: Programming language model organized around objects rather than "actions" and data rather than logics

UML: Diagram based on the UML (Unified Modeling language) with the purpose of visually representing a system along with its main actors, role.

Intersection: An area shared by two or more roads.

Tile: a smallest countable unit on the City Boost simulation map.

GUI: Graphical User Interface.

Time thread: the smallest sequence of programmed instruction that can be managed independently by a scheduler.

Turn: Action that each car agents will take at the intersection

1.4 References

https://searchmicroservices.techtarget.com/definition/object-oriented-programming-OOP
https://tallyfy.com/uml-diagram/

1.5 Overview

The hope is that the NetFlow Traffic Simulator System will become an efficient, portable, and scalable tool for more efficient city design that can be hosted online which would allow this resource to be shared with other cities as well.

2 Current Software Architecture

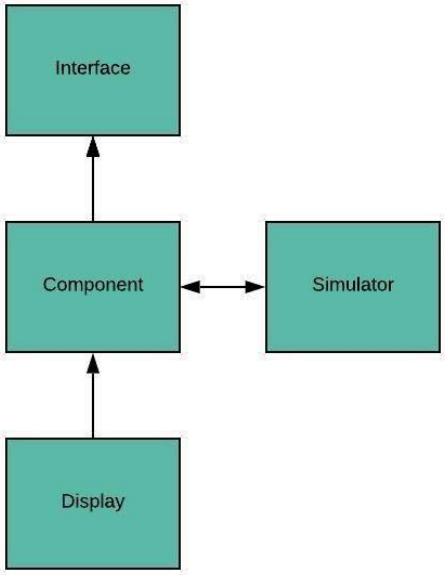
3. Proposed Software Architecture

3.1 Overview

The proposed system incorporates the Model, View, Controller design pattern. The User (Mayor Mann) interacts through the console (Controller) while the Map object with its individual traffic components (Model) contains the relevant data. The data and its resulting statistics from the model can be viewed via the Report class (View).

3.2 Subsystem decomposition

The proposed system is broken down into 4 parts: Interface, Display, Component, and Simulator. The architecture model is shown graphically below.



System Model-view-control architecture

The Interface subsystem will be able to load the map data, and get the initial statistic query of specific objects which comes from user inputs. So, the two main purposes for this subsystem are reading data from user inputs and provide an interface to load map data and manipulate traffic components. The publicly-accessible methods call from Component Subsystem to Interface subsystem is in SimulationMap class: funcion “loadComponents()”, “loadCars()” and “loadTiles()”. These three functions will use the user provided json file to read the information of map, cars and components. And Component Subsystem is responsible to save these information.

The Component subsystem will be able to store the position of multiple cars moving on the map.

The purpose of Simulator subsystem is to control events. It will recognize different conditions that may affect moving cars, get changes under these different events, and finally be able to update the changes of cars position to Component subsystem. Simulator Subsystem works with Component Subsystem together. Component Subsystem will be able to store user provided informations by the “load...” functions, and Simulator Subsystem should be able to read the command from Interface to decide start/ stop/ restart the simulation by the function “getConsole()”.

The Display subsystem will be able to show live statistic report of each car, and live map with moving cars. It will also be able to generate final report of running time when all iterations end.

Display Subsystem will interact with Component Subsystem. Display is able to show a live map with moving cars and state changing intersections. By function “displayMap()” in Display, this subsystem can get all information from Component Subsystem.

The architecture we chose is MVC, Model, View, Controller. Because our application can be simply separate into these parts. Interface will be the Controller, where it accepts the data from user input. Component and Simulator will work as the Model, where do the management of data. Display subsystem is View to show final report. The advantages to do this way are, first, it will be low coupling. For example, even we changed the input from users in Model (Interface), the way we manage our data in Component and Simulator will still be same. Second, because of low coupling, it makes easier to for our team to code these subsystems simultaneously.

3.3 Hardware/software mapping

Our program should work on any platform supporting JAVA. There's no hardware requirement for our program.

Input/ Output Performance:

Rate to update the map will depend on the number of traffic components. Suppose the user set more cars moving on the map at the same time, the program will take longer to update the map.

Processor Allocation:

The Interface and Display subsystems will take place on the user's workstation. Simulation and Component will be able to operate on any machines provided.

3.4 Persistent data management

Our system will store all pixel of the map, cars, and traffic component information. The memory size will be dependent on number of cars, number of traffic components, and size of the map from user inputs.

3.5 Access control and security

The design goal for our project was to run multiple simulations of traffic flows and get the best combinations of traffic components. There's no potential security issue we must take care about.

3.6 Global software control

The initiated requests of our program will be the user loading his CSV file of traffic component combinations and car information. The connection between Component and Simulator subsystems will take care about synchronized objects. The changes of every traffic component will be handled in Simulator and update synchronously and store in Simulator. The system should follow the synchronous sequence, started from Interface, and then go to Component, which also interact with Simulator, finally go to Display subsystem.

3.7 Boundary conditions

The start-up boundary is whenever the user loads his CSV file with the information about number of cars, position of cars, and combination of traffic lights and stop signs.

The shutdown boundary is whenever the user decided to press stop and change combinations of traffic lights and stop signs. It will restart our program then.

The final shutdown boundary is when every car reaches the destination and system shows final report of traffic flow.

Error behaviors are whenever users have invalid inputs, for example, the traffic component is out of map.

4. Subsystem Services

The Interface subsystem will provide traffic components information from user input. The Simulator subsystem will provide how these traffic components get change under different conditions. The Component subsystem depends on these two subsystems will be able to store the information of traffic components, and then offer these to Display subsystem. Display subsystem will finally generate the live statistic report based on the data we got from the Component subsystem.

5. Class Interfaces

For more detail on Class interface API please refer here. (https://datduyng.github.io/cityboost/doc-java/index.html)

Interface Subsystem:

Class	Description
Console	This Abstract class provide user the necessary interface to the Simulator.
CLI	CLI(Command Line Interface) is an implementation of Console. This class provide user with command line interface with the simulator
GUI	GUI(Graphic user interface) is an implementation of Console. This class provide user with graphic user interface with the simulator

Console Method Summary:

Modifier and Type	Method and Description
Void	DisplayHelp() Display user interface help
Void	Run(String command) Run an interactive command

Component Subsystem:

Class	Description
Point	Point class is used to show current position
Car	Car class is used to serve as an agent that interact in the simulator
SimulationMap	Map class that have all components
Ground	Ground class is an implementation of Tile class that specifies a ground tile
Road	Road class is an implementation of Tile class that specifies a road tile
TrafficLight	TrafficLight class is an implementation of the abstract Intersection specifies a trafficlight intersection
StopSign	StopSign class is an implementation of the abstract Intersection specifies a StopSign intersection
Interface	**Description**
Tile	Tile Abstract Class is a smallest countable unit that make up a Map in the simulation
Intersection	Intersection abstract class is an implementation of Tile class that specifies a intersection tile

Car constructors:

Modifier	Constructor and Description
String	Id The representation of different cars
Point	Start Start position of cars
Point	Stop Stop position of cars
Point	currentPosition Current position of cars
Int	CurrentSpeed Current speed of cars
String	Direction The moving direction of cars: “>”, “<”, “^”, “v”
Int	Increment

	The small moving increments in each tile of every 0.5 second
String	State The current state of cars: “moving”, “passing”, “stopped”, “deadend”

Car method summary:

Modifier and Type	**Method and Description**
Void	Move() Car moving method
Void	Stop() Change car’s state to stop to stop car moving
Void	Waiting() Change car’s state to stop to stop car moving (used for the first 0.5 second)
Point	MoveTile() update car’s tile position based on the direction
Boolean	CheckTile() Check next tile based on car’s direction
String	Turn() Turn method for cars used on intersection
String	GetAttr() return necessary report information for Report Interface to be displayed

Point constructors:

Modifier	**Constructor and Description**
Int	X X value in 2D matrix map
Int	Y Y value in 2D matrix map

SimulationMap constructors:

Modifier	**Constructor and Description**
private int	pixelSize size of smallest countable unit on the simulation map
protected Arraylist	carList keeps track of all agent in the simulator
Protected ArrayList	componentList Keep track of all traffic component object in the simulator
private int	sizePix account for distance in mph mapping to the simulation
private int[][]	layout

	2d Matrix keeps track of all tile represent the whole simulation map

SimulationMap method summary:

Modifier and Type	Method and Description
public	Map(int pixelSize, int numWidth, int numHeight) Constructs a Map objects that represent all subunit of the simulation. Given pixelSize, number of pixel in x and y -axis.
Void	loadComponent(String filePath) load all traffic component given a '.JSON' file with all component info such as tile type, car information, and traffic components information
Void	loadCar(String filePath) load all cars given a '.JSON'
Void	loadTiles(String filePath) load all tiles (could be ground, road, or intersections) given a '.JSON'

Tile (Abstract class) constructors:

Modifier	Constructor and Description
String	Type Type of this tile: could be ground, road, or intersections
Point	MapIndex
Point	Position Tile position on the map

Tile (Abstract class) method summary:

Modifier and Type	Method and Description
public	GetNumCarPass() Used to get the order of cars passed this tile

Ground constructors:

Modifier	Constructor and Description
String	Type Type of this tile: Ground

Ground method summary:

Modifier and Type	Method and Description
Public	GetNumCarPass() Used to get the order of cars passed this tile

Road constructors:

Modifier	Constructor and Description
String	Type Type of this tile: Road

Road method summary:

Modifier and Type	Method and Description
Public	GetNumCarPass() Used to get the order of cars passed this tile

Intersection (Abstract class) constructors:

Modifier	Constructor and Description
String	Type Type of this tile: Intersection
Int	Increment Increment of intersection will keep track of the color of Traffic Light or the stop seconds of Stop Sign
String	BuiltDirections

Intersection (Abstract class) method summary:

Modifier and Type	Method and Description
Int	UpdateIncrement() Update the increment to keep track of the changing state of Traffic Light or Stop Sign
String	GetContent() Get the content of intersection is Traffic Light or Stop Sign
String	CarEnter() Used to get the order of cars passed this tile

TrafficLight constructors:

Modifier	Method and Description
String	Type Type of this tile: TrafficLight
String	Color Color will change from green to red with the change of time
PriorityQueue<Car>	NsTraffic It represents north-south Traffic Light
PriorityQueue<Car>	EwTraffic It represents east-weatTraffic Light

TrafficLight method summary:

Modifier and Type	Method and Description
Int	UpdateIncrement() keep track of the color changinng
Void	SwitchColor Used to change color with the change of time
Void	EnQueue Adds car to appropriate queue based on direction.
Void	Dequeue Removes car (if any) from appropriate queue based on current traffic-light direction, resets its state to 'stopped'

StopSign constructors:

Modifier	Method and Description
PriorityQueue<Car>	CarEnter
String	State

StopSign method summary:

Modifier and Type	Method and Description
Int	UpdateIncrement() Update the increment to keep track of the changing state of Traffic Light or Stop Sign
Boolean	AddCarToQueue boolean true if car is successfully added
Void	DeQueue Get Car that is at the head of the priority queue

Simulator Subsystem:

Class	Description
Simulator	Program that is intended to be running on it owns. This class initialize a full traffic simulation

Simulator Method Summary

Modifier	Constructor and Description
Void	StartSimulation() Start a timer and events in simulation
Void	restartSimulation()

	Restart all field in simulation environment
Void	stopSimulation() stop timer and events in simulation
Console	getConsole(String id) Find and return the Console in the user environment represented by this simulator context with a given id
Void	saveSimulation(String filename) save all field and status of a simulation to a json format Parameters: filename: filename that one would like to save the simulation status at.
Void	loadSimulation(String filename) load previously saved field and simulation status Parameters: filename: file path to the previously saved status

Display Subsystem:

Class	**Description**
Display	Display class provide visualization for all TrafficComponent classes
Interface	**Description**
Report	Simple abstraction for display summary report of all traffic component.

Display Method Summary

Modifier	**Constructor and Description**
Void	DisplayMap() Display map to the screen
Void	DisplayTrafficComponent(String command) Display and update the simulation on window

Report Method Summary

Modifier	**Constructor and Description**
Simulator	GetSimulator(String name) Find and return the Simulator in the user environment represented by this simulator context with a given name
Int	GetTotalTime() return the current timer of the simulation
String	FinalReport() Return information across all field in the simulator environment

6. Detailed Design

UML Class diagram:

Abstract class Console will be the interface to run user command, and it implements CLI (Command Line Interface) and GUI (Graphic Line Interface) classes.

Car class will have basic information of every car component.

One Map will have multiple Tiles, which is an abstract class. A tile could be either ground, road, or intersections (traffic light or stop sign).

Interface report will show the statistic report of each car running time and final statistic report based on different combinations of traffic components. The Display class will show live map and traffic object on map.

Simulator will be our main class. It will be the event controller to deal with the change to every cars under different conditions.

Car, Simulator, Console, Display and Report will be associated with Map. Map has a public attribute to get carList, and it will be shared to every other associated classes.

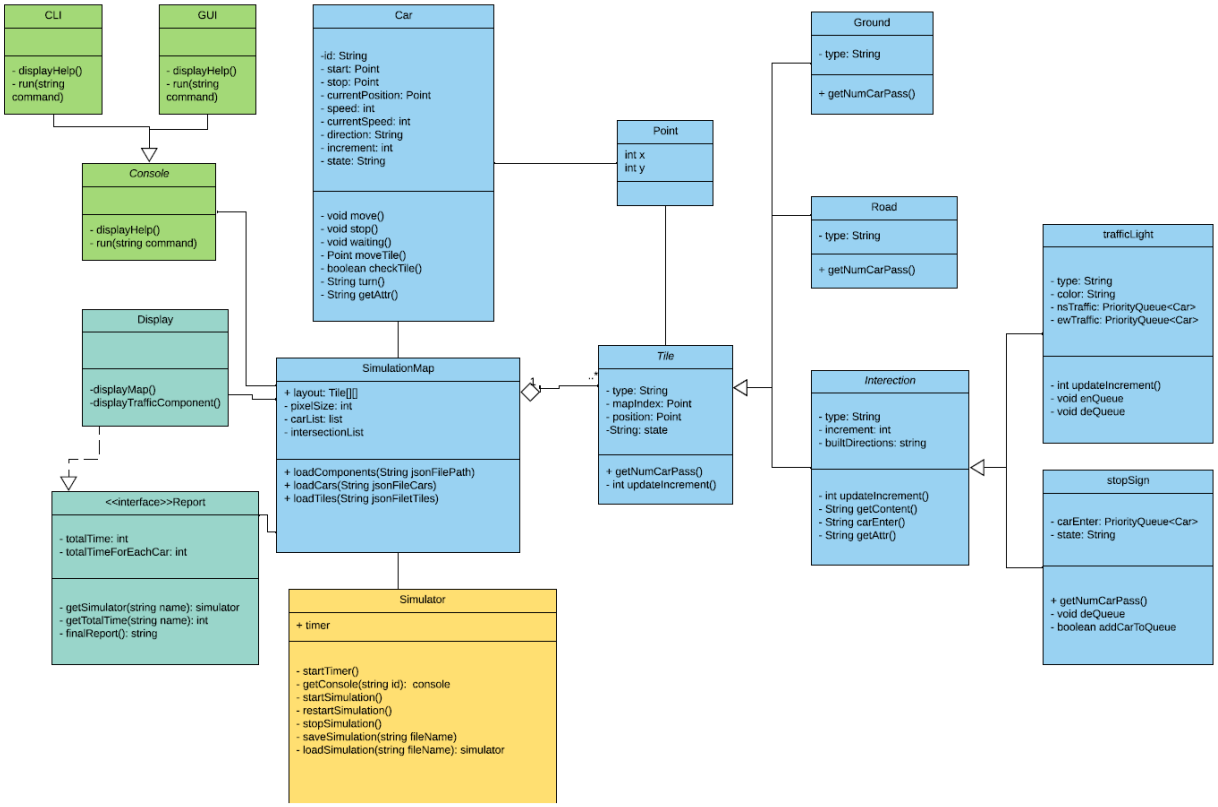
Edited Class diagram to portray structural view of the solution domain:

Green represents Interface subsystem;

Blue represents Component subsystem;

Yellow represents Simulator subsystem;

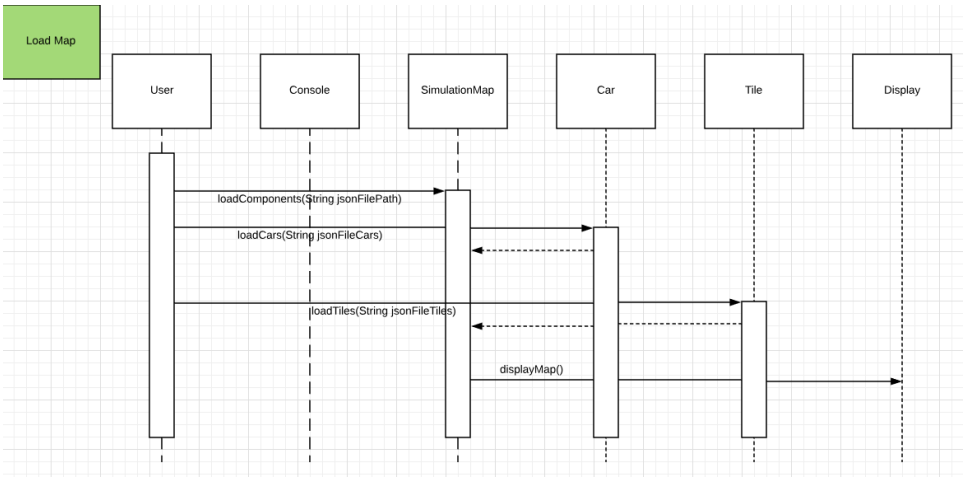
Aqua represents Display subsystem.



Sequence Diagram:

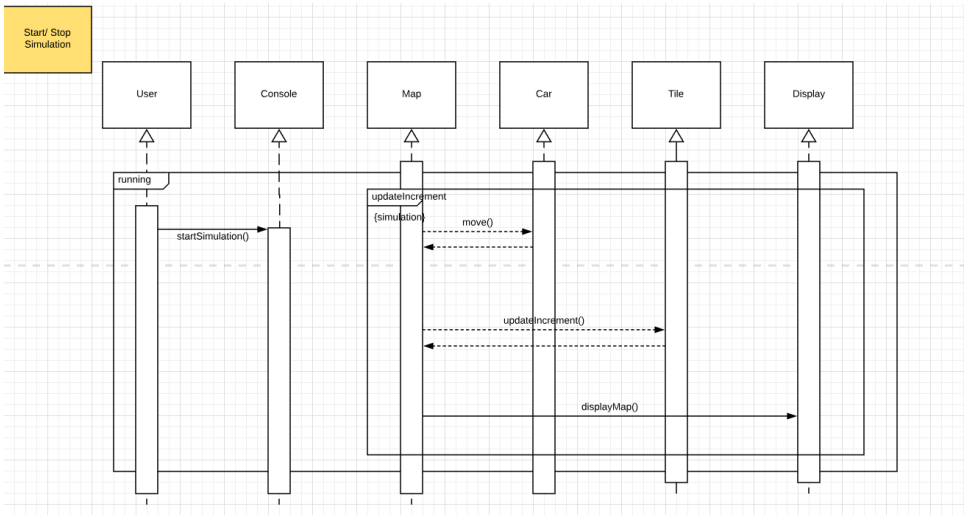
Use Case: Load Map

To load the map along with its traffic components, the User Mayor Mann first makes a call to the function, 'loadComponent(String filePath)'. Next, he loads the cars and traffic components via their json files. These calls return the car and component objects back to the Map object, which then can display the loaded map to the user.



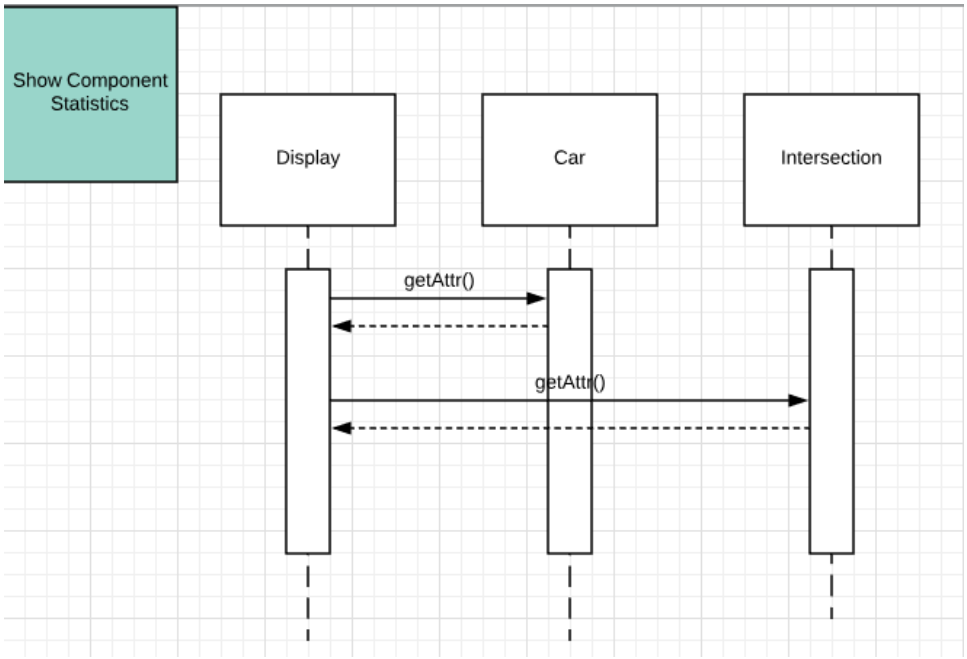
Use Case: Start/Stop Simulation

The user chooses to start the simulation by either pressing a button or entering a command via the console which enters the ‘running’ loop and calls ‘startSimulation()’. This starts incrementing the timer/counter which will be used to determine the position of all cars on the map and the timing for the traffic components.



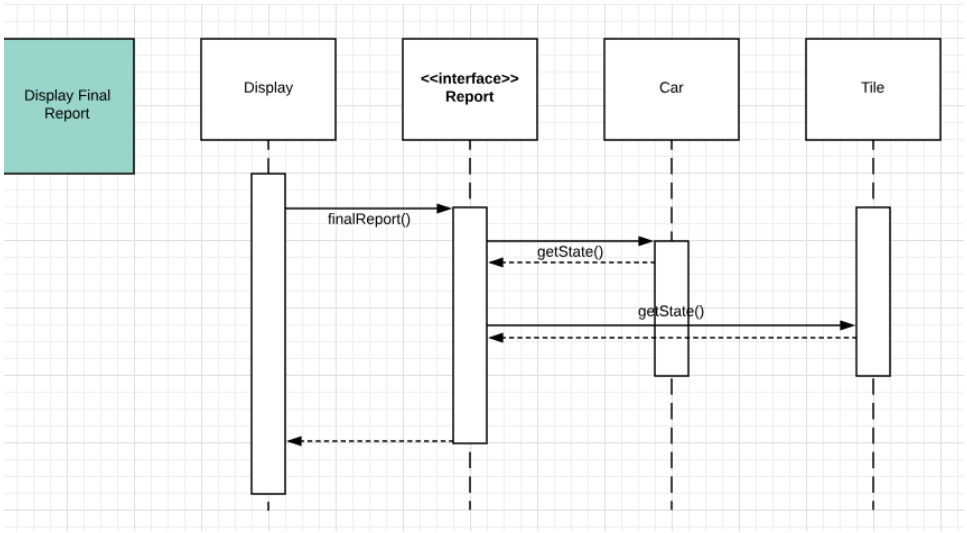
Use Case: Show Component Statistics

To show live statistics, the Display class will make calls to the ‘getAttr()’ function of both Car and Intersection (Traffic Component) respectively and display the results to the User.



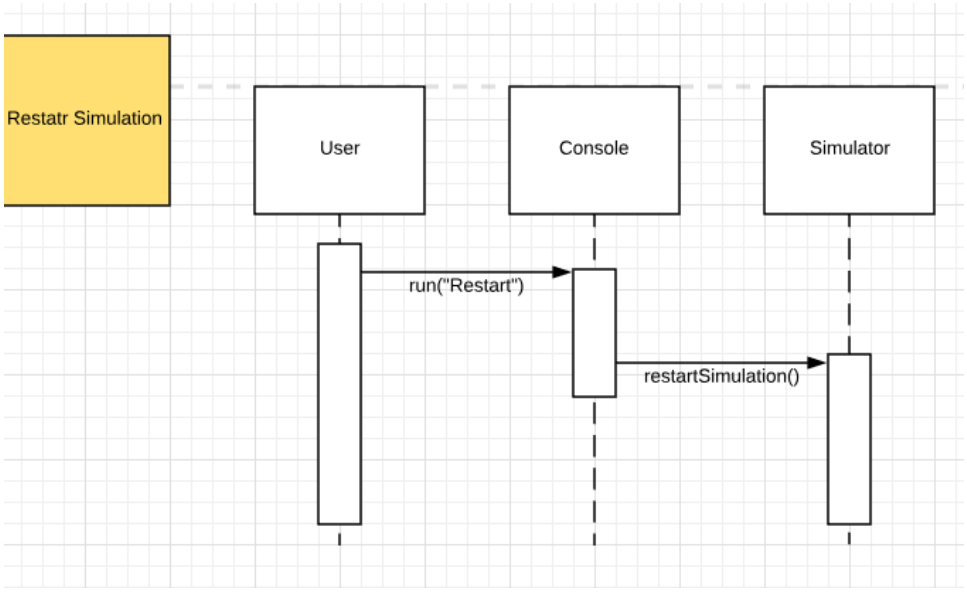
Use Case: Display Final Report

To display the final report, the display calls finalReport from the Report Interface which retrieves the car and traffic component statistics via ‘getCarState()’ and ‘getComponentState()’ functions respectively. The returned attributes to the Report Interface are coalesced and returned as a string to the Display.



Use Case: Display report

Once all cars reach their destination it is considered over. The user can also choose to end the simulation early. To restart a simulation, the User either presses a restart button or enters a restart command via the console. This will call ‘restartSimulation()’ function in Simulator.



Use Case: Restart simulation.

7. Glossary