City Traffic Simulator

Software Design Document

Team 14
Mike Senior
Cal Leising
Louis Shakya
Nathan Luchsinger

Introduction

Purpose

This system aims to simulate the flow of traffic in the city of Pacopolis. Specifically, it will allow a user to change the arrangement of traffic lights and stop signs at intersections, then simulate the flow of traffic so the user can see the affect the arrangement has on how quickly the cars reach their destination. Using this information, a user can determine the optimal layout that will get the most cars to their destinations the fastest.

Design Goals

Our overall design goal is to utilize OOP principles to create a robust and easily-maintained system. In order to do this, we will look to create a software architecture that allows for high cohesion in our system, while also minimizing coupling in our system. This will make our system easier to maintain in the future. We also want to ensure all aspects of our system traceable. In order to do this, we will utilize our Requirements Analysis Document from previous increments, as well as the City Traffic Simulator problem statement. Additionally, we will follow specific interface documentation guidelines and coding conventions. They are: 1) Classes are named with singular nouns, 2) Methods are named with verb phrases, 3) Fields and parameters are named with noun phrases, 4) Error status is returned via an exception, 5) Any collections and containers have an elements() method returning an Enumerations, and it will be robust such that Enumerations will be correct despite the removal of elements, 6) Utilize comments to improve readability. These guidelines and conventions will make it easier to keep interface design consistent. Also, we look to make our system as user-friendly as possible.

Design Trade-Offs

The largest design trade-off we made regarded the decision to store the times recorded by the system as flat files on the users machine, rather than store them in a separate database. This decision was done to simplify the design of the system at the expense of a larger storage footprint to store the data. However, we feel it won't require much more space to store this information, making the trade-off of design simplification worth it. Additionally, we feel this design decision will increase the response time of our system. Another design trade-off we have made regards functionality and usability. We decided to attempt to make our system as easy-to-use as

possible. As such, the user is responsible only for inputting start and end points for cars, as well as changing intersections. We feel this allows our system to be very easy to use, while still maintaining all necessary functionality.

Definitions, Acronyms, and Abbreviations

Cohesion - the strength of the dependencies within a subsystem or class

Coupling - the strength of the dependencies between two subsystems or two classes

Exception - an unexpected event that happens during the execution of the system

Maintainability - ability to change the system in the future

OOP - Object-Oriented Programming

Readability - how easily a system can be understood from its source code

Subsystem - a smaller, simpler part of the larger system

Traceability - whether an aspect of the system can be traced back to the original requirement or rationale that motivated its existence

References

Requirements Analysis Document for Team 14 from Increment 1 and 2

Increment 3 directions from Dr. Bohn's file Increment 3.pdf on Canvas

City Traffic Simulator problem statement from increment 1 section 6

Object-Oriented Software Engineering Using UML, Patterns, and Java, 3rd Edition, by Bernd Bruegge and Allen H. Dutoit

Overview

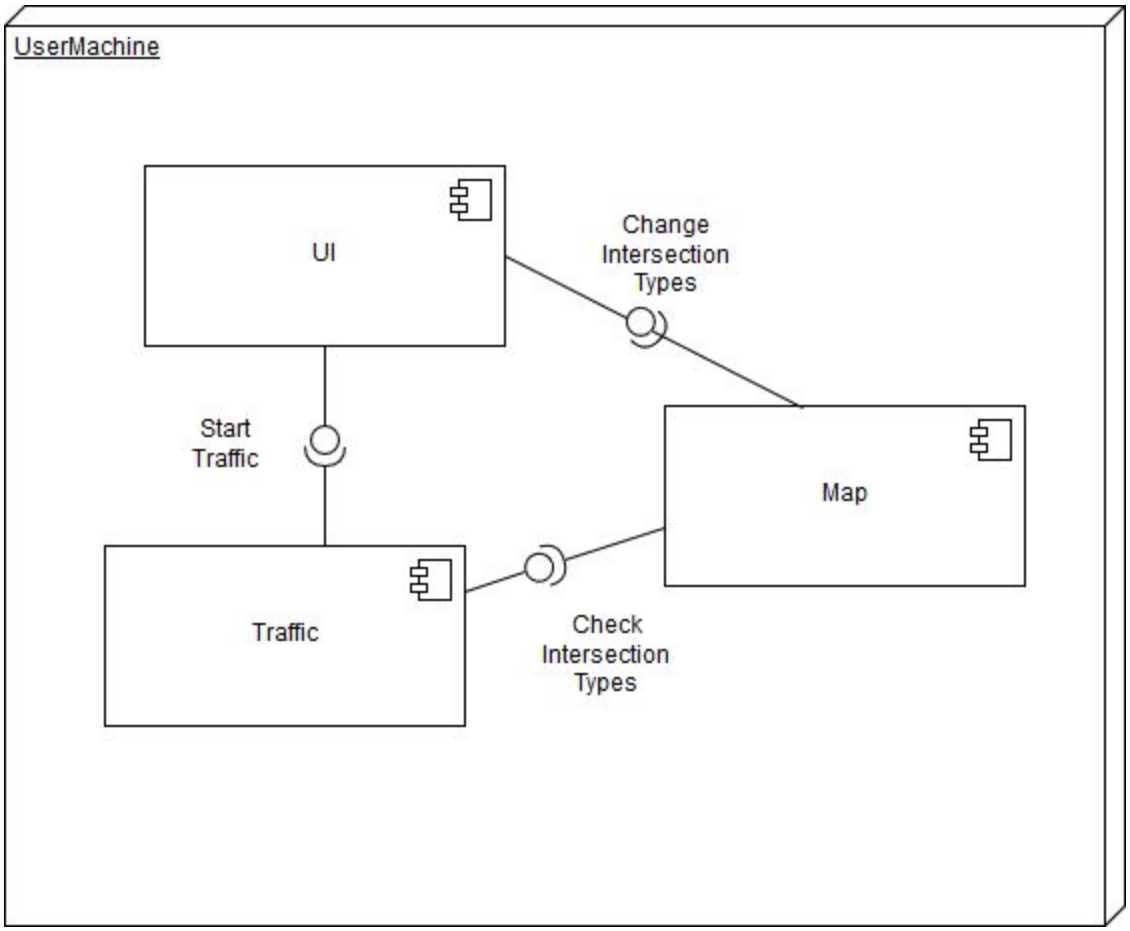
In this document, we look to detail the design decisions we made regarding our system. In particular, we will discuss our proposed software architecture and the subsystems we have divided our system into. We will talk about the background information we used to design our system, and the common issues we address with our system. Next, we detail our proposed software architecture. This section includes an architecture diagram to make it easy to visualize our system, as well as a brief reasoning for our design decisions and the responsibilities of each subsystem. Also, we include a discussion of the operations performed by each subsystem. Then, we describe the classes and their public interfaces, followed by design-level class and sequence diagrams. Finally, we include any important terms and their definitions in a glossary.

Current Software Architecture

Overview

For the software architecture the classes have been split into the following subsystems: Timer, Map, Traffic, and UI. UI manages the user Interface and reads inputs from the user. Traffic manages the cars and their interaction with intersections. Map manages the intersections, starting points, and destinations. We based our design loosely off of the Model-View-Controller architectural style.

Subsystem decomposition



Because the functionality of our program is only on one machine and does not use outside servers we do not have to worry about interactions between the user, the program, and a server. We used this architecture in order to have the map, traffic, and UI effectively work together while having the timer read the results.

The UI allows the user to set what type of intersection each intersection will be and and allows the user to start the traffic simulation. The Map keeps track of all the types of intersections and keeps track of when the lights for intersections should switch from green to red. Traffic keeps track of all the cars on the maps, manages how cars interact with each kind of intersection, guides cars to their destination, and checks if all the cars have reach their destination.

Hardware/software mapping

For our system, all subsystems are pieces of software that will run on the machine of the user. Due to this, no issues are introduced by multiple nodes, as the entire system is run on one hardware node. This allows us to keep the system simple, while still allowing us to have all necessary functionality. Additionally, we don't plan on reusing any existing software. This creates no issues, but it will require us to create more software to complete the system.

Persistent data management

One example of persistent data we have to manage in our system are the times it takes all the cars to reach their destinations. In order to track this, we will record the times in a list, and allow the user to view these times. This list will be stored in the Timer subsystem. Upon the termination of the system, the times that have been stored in the list will be added to a file containing the times that have previously been generated by the system, or, if no such file exists, one will be created to store the data. This will allow the timer information to persist across multiple uses of our system, allowing the user to easily reference this information to make decisions about what layout to use.

Access control and security

For our system, all users will have access to the same functionality of the system. Any user will be able to upload location data for cars, change the types of intersections, and start and reset the simulation, no special privileges will need to be granted. Also, the only data that is stored will be stored locally on the users machine, and accessing this file will be done by the user Due to this, access control and security aren't really applicable to our system.

Global software control

Our software control method is centralized control, as the user will choose an option from the UI subsystem, then functions from other subsystems will be called in order to perform the task the user requested, before control is eventually returned to the UI subsystem, which will allow the user to further interact with the system. The most difficult issue regarding synchronization and concurrency is allowing all cars to move simultaneously without any "collisions" occurring. This has been simplified by ensuring all drivers are perfect, meaning they will always avoid a collision. In order to implement this, our Car objects will have a certain length, and we will ensure the lengths of cars never overlap. The handling of this issue will be performed in the Traffic subsystem. When the user selects an option from the UI subsystem, it will request that the Timer, Map, and Traffic subsystems perform the necessary functions to carry out the user requests. The Traffic subsystem will manage the synchronization of moving the cars, requesting the type of intersection ahead from the Map subsystem for each Car object as it approaches an intersection. Additionally, the Traffic subsystem will request the Timer subsystem stop the timer and report the final time.

Boundary conditions

Upon starting up the system, the user will be required to upload the start points and the destinations of all the cars they want to test, as these won't be stored after the user shuts down the system. The map will be same every time, and will automatically be generated when starting up the system. Also, all intersections will be created, and the type of intersection will automatically be set as a traffic light. The user will be required to manually change the type of an intersection after starting up the system if they don't want it to have a traffic light. When the system is

shutdown, the timer data that is stored in the timer subsystem will be appended or written to a file, depending on if a file to store such data exists already. No subsystem will be allowed to terminate independently, the entire system must be terminated. Also, since there is no database, no updates need to be communicated upon termination. The majority of errors in the system will occur when the user inputs the start locations and destinations of the cars. In order to ensure this doesn't break the system, the start points and destinations of each car will be checked to ensure they represent a valid location within the bounds of the map.

Proposed Software Architecture

No change

Subsystem Services

UI Subsystem: This subsystem will only consume services offered by other subsystems. This is the "View" subsystem of our modified MVC architecture, and as such, only displays application domain objects, and allows the user to interact with other subsystems.

Map Subsystem:

changeIntersection() - this method allows other subsytems, namely the UI subsystem, to change the type of a particular intersection when the user chooses to do so.

getIntersection() - this method is called by other subsystems to determine what type of intersection a certain intersection is. If it is called by the UI subsystem, it will be used to display the type of intersection to the user on the user interface. If it is called by the Traffic subsystem, it is used to determine how a car object should proceed as it nears an intersection.

Traffic Subsystem:

startTraffic() - this method causes the car objects to begin moving towards their final destinations

resetTraffic() - this method returns all car objects to their starting points, and then allows the start and end points of cars to be modified

Class Interfaces

User Interface Class:

This class is responsible for displaying a model of the system to the user, as well as allowing the user to interact with the system through it. It offers no services to other classes, it only consumes services from other classes in order to perform the requests of the user. As such, it has very few public fields and methods: it has a public method, main(), that is the main method for our system. Additionally, it has the public field checkboxRet[], which is an array that keeps track of the intersections on the map, and allows the system to keep track of which intersection is changed by the user, and updates it accordingly.

Intersection Class:

This class is responsible for modeling the intersections on the Pacopolis map. It allows the intersection to be changed by a different type when the user changes one on the GUI, and it also allows other classes, such as the Car

class, to determine what type of intersection it is approaching, and how the Car object should behave as it approaches the intersection. As such, it has 1 public method, changeIntersection(), which is responsible for changing the type of intersection to one that a user chooses.

Map Class:

The Map class is responsible for ensuring the traffic follows the layout of the Pacopolis map, and that all cars obey the rules at an intersection. This class will hold a list of the intersections, so they can be checked by the getIntersection() method, so it will need to interact with the Intersection class. Additionally, it will need to interact with the Car class, so the Car objects know what kind of intersection they are approaching. This class contains the public method getIntersection(), which will be used by the car class to determine what kind of intersection it is approaching, so it may proceed accordingly.

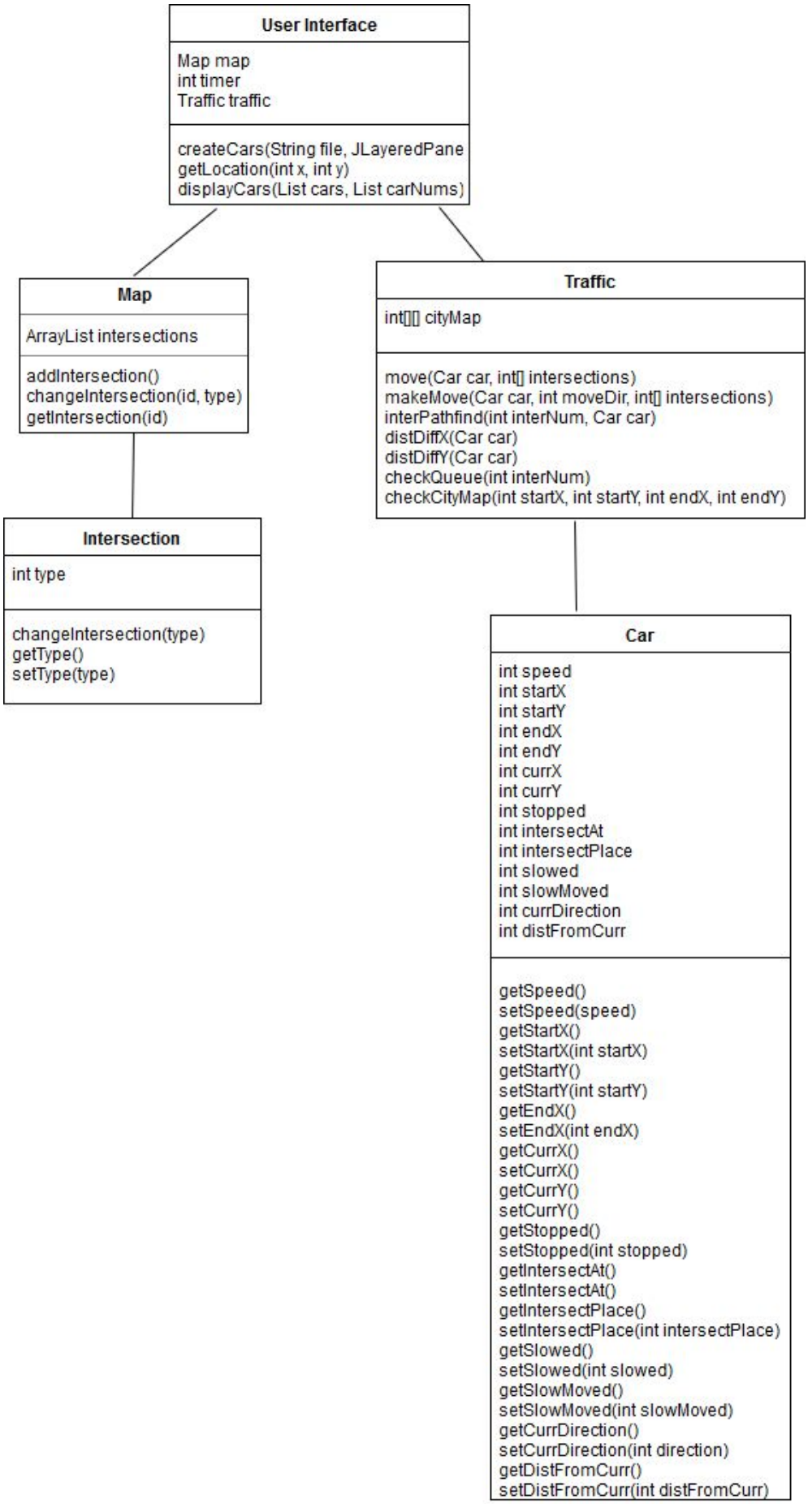
Car Class:

The Car class will be used to model the cars on the roads of Pacopolis. This class will interact with the Map class to determine whether it is nearing an intersection, and what kind of intersection it is approaching, so it may proceed according to the rules of that intersection. Additionally, it must interact with the Traffic class, as the Traffic class keeps track of how many cars have arrived, and stops the simulation when all cars have reached their destinations. As such, the Car class has a public value arrived that is a boolean, and will be true when the car has arrived, and false if it hasn't arrived yet. Additionally, it has the public methods start(), stop(), moveTowardsEnd(), and reset(), so that the cars can start at the beginning of the simulation, as well as start and stop for traffic lights and stop signs. The reset() method is used to reset the cars to their starting locations when the user presses the reset button.

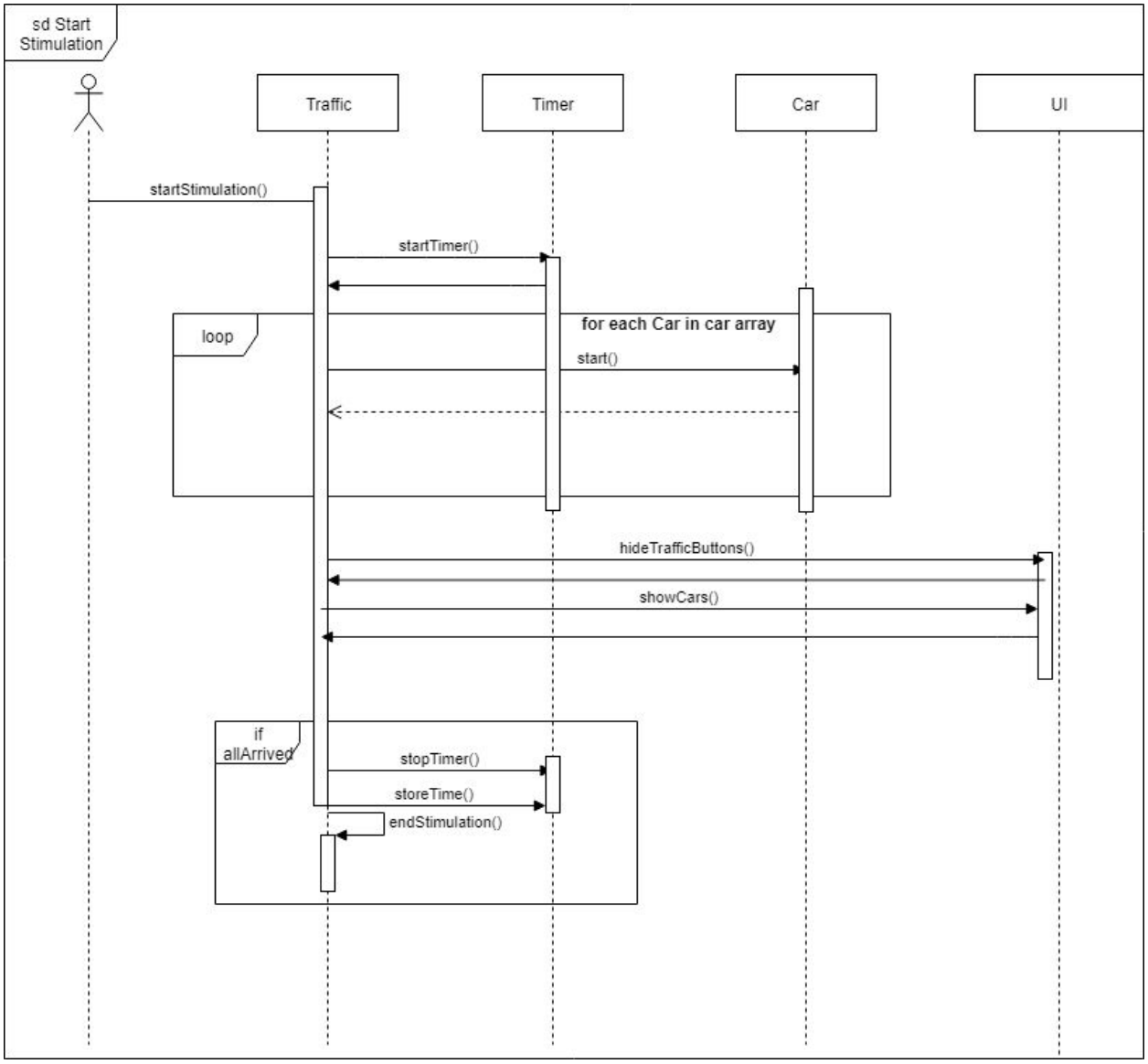
Traffic Class:

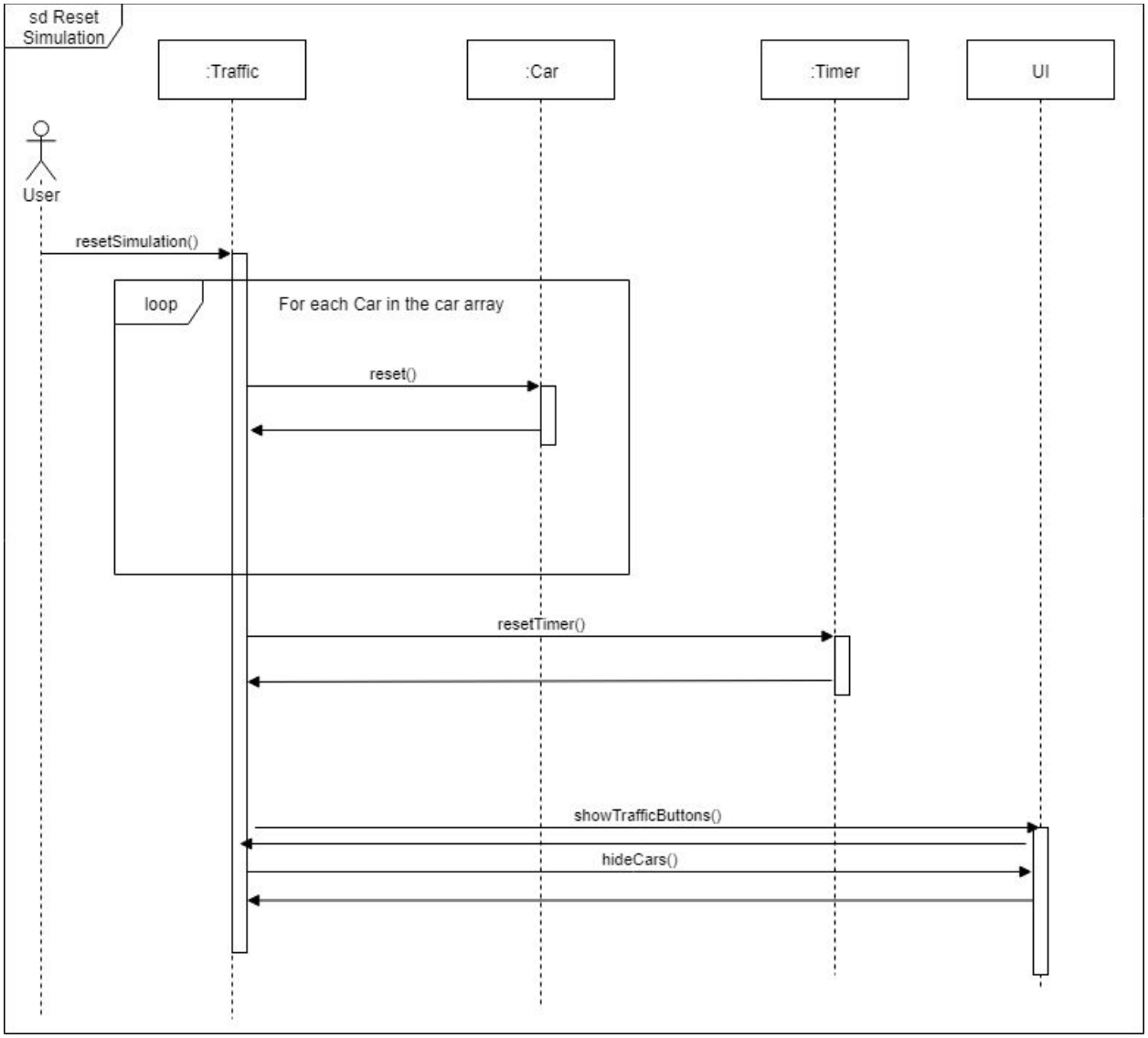
The Traffic class will be used to ensure traffic in the simulation flows smoothly. This class uses methods provided by both the Car class and the Map class in order to fulfill its duty of ensuring traffic flows smoothly. Additionally, the User Interface class communicates with the Traffic class through the public methods when the start, reset, or stop button has been pressed. It has a private allArrived field that will be updated to true when all cars have reached their destination, which will end the simulation. Additionally, it contains a private list of Car objects, so it can control the flow of cars on the road via their public methods. It contains three public methods, endTraffic(), which will cause all cars to stop if the user stops the simulation, startTraffic(), which will cause cars to embark toward their final destinations when the simulation is started, and the resetTraffic() method, which causes all cars to return to their start points.

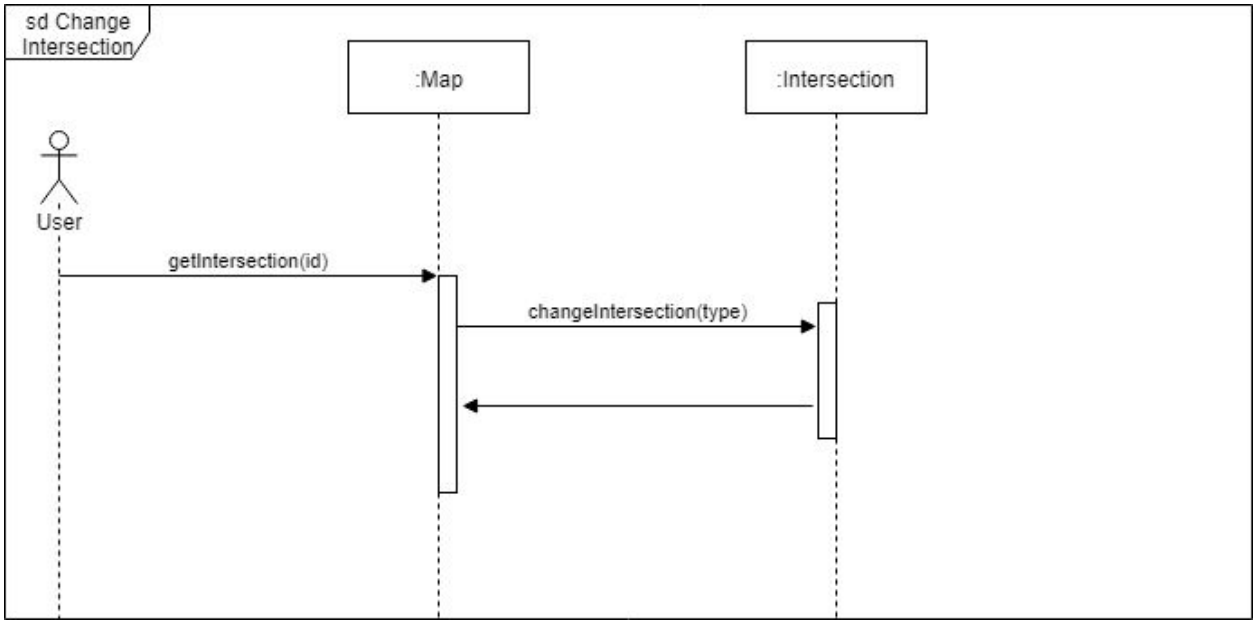
Detailed Design


User Interface
Map map
int timer
Traffic traffic
createCars(String file, JLayeredPane
getLocation(int x, int y)
displayCars(List cars, List carNums)
Map
ArrayList intersections
addIntersection()
changeIntersection(id, type)
getIntersection(id)
Traffic
int[][] cityMap
move(Car car, int[] intersections)
makeMove(Car car, int moveDir, int[] intersections)
interPathfind(int interNum, Car car)
distDiffX(Car car)
distDiffY(Car car)
checkQueue(int interNum)
checkCityMap(int startX, int startY, int endX, int endY)
Intersection
int type
changeIntersection(type)
getType()
setType(type)
Car
int speed
int startX
int startY
int endX
int endY
int currX
int currY
int stopped
int intersectAt
int intersectPlace
int slowed
int slowMoved
int currDirection
int distFromCurr
getSpeed()
setSpeed(speed)
getStartX()
setStartX(int startX)
getStartY()
setStartY(int startY)
getEndX()
setEndX(int endX)
getCurrX()
setCurrX()
getCurrY()
setCurrY()
getStopped()
setStopped(int stopped)
getIntersectAt()
setIntersectAt()
getIntersectPlace()
setIntersectPlace(int intersectPlace)
getSlowed()
setSlowed(int slowed)
getSlowMoved()
setSlowMoved(int slowMoved)
getCurrDirection()
setCurrDirection(int direction)
getDistFromCurr()
setDistFromCurr(int distFromCurr)

Implementing the new sensor-based traffic light didn't require any large changes to our classes. We had to add an additional type of intersection, and modify our getType() function accordingly. Additionally, we had to implement some logic in the Traffic and UserInterface classes in order to properly change the sensor-based intersections, and correctly display the sensor-based traffic light on the map to the user.




sd Reset Simulation
:Traffic
:Car
:Timer
UI
User
resetSimulation()
loop
For each Car in the car array
reset()
resetTimer()
showTrafficButtons()
hideCars()



Glossary

Cohesion - the strength of the dependencies within a subsystem or class

Coupling - the strength of the dependencies between two subsystems or two classes

Exception - an unexpected event that happens during the execution of the system

Maintainability - ability to change the system in the future

Readability - how easily a system can be understood from its source code

Subsystem - a smaller, simpler part of the larger system

Traceability - whether an aspect of the system can be traced back to the original requirement or rationale that motivated its existence