City Traffic Simulator

Software Design Document

Team Name
Chavez, Bryan
Majid, Mohammad
Nguyen, Brian
Vuong Nguyen, Huy

Introduction

Purpose

This purpose of this document is to provide detailed description of the design of a system capable enough to allow for software development to continue with an understanding of what is to be built and how it is expected to built.

Design Goals

The system will make use of Object Oriented Design Paradigm. The system will be able to stop, start, and reset/quit the system. Objects will be set at their appropriate locations. The system should be able to stop cars at traffic lights for two minutes. Cars will also stop at stop signs for a second. After the cars start accelerating it should only accelerate up to 15 mile per hour for the first 1/8th of a mile then after that it should continue up to 30 mph and maintain the speed from there.

Design Trade-Offs

The system platform will be relatively generic which helps readability and reusability.

1. Datas are imported and exported in CSV instead of database:
    - Easy to use, easy to read, user friendly
    - Hard to maintain when we have arbitrarily large amount of csv files. Cost a lot of storages.
2. Software is developed using OOP paradigm:
    - Reusability, we basically create our own API.
    - Easy to implement, deploy, test and debug.

Definitions, Acronyms, and Abbreviations

OOP: Object-oriented programming

GUI: Graphical user interface.

UML : Unified Modelling Language

Python : General programming language

BDD : Behavioral driven development

XML : Extensible Markup Language

References

https://techterms.com/

Overview

The overview of this project is mainly organizing enough traffic lights and stop signs to be processed correctly and set in the city of Pacopolis. The city of Pacopolis' goal is achieving the necessary help in setting the traffic lights and stop signs to help minimize the amount of time needed for each vehicle to get to their specific destination. This system will eventually minimize accidents and damages done in the city too eventually becoming a functional city that will help portray a correct way of driving getting to the needed destination in less time.

Current Software Architecture

Proposed Software Architecture

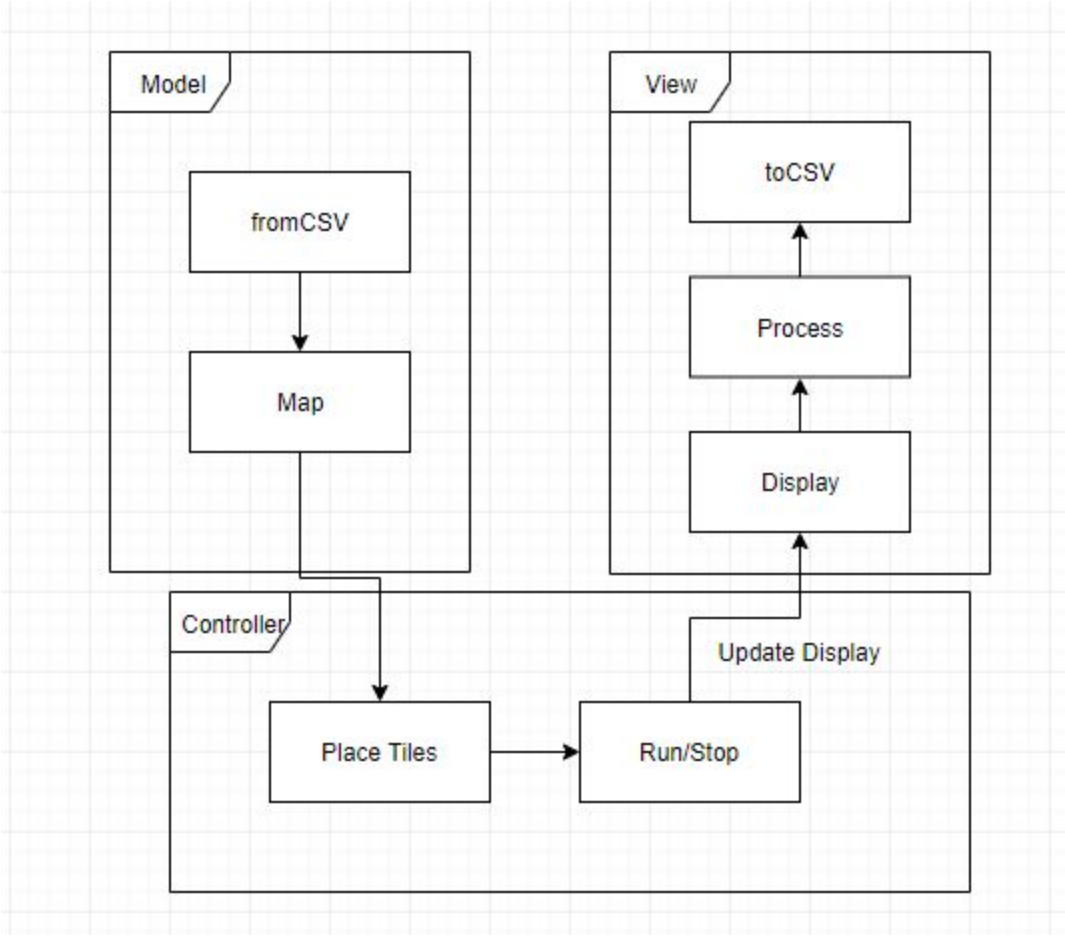
Overview

We will be using the Model View Controller architecture for our system. We will be explaining the software design of our project. These software designs will include Subsystem Decomposition where it explains all the subsystems that are part of the simulation and their jobs. It also explains where the data is being stored with the explanation of Access Control and Security. Global Software Control will be explained as well which will explain how the system is being created and how it works. It will also show how the subsystems interact to achieve the goals. Subsystem services will give you the summary of the subsystems and their jobs. API of each subsystem is also shown and explained.

Subsystem decomposition

We have multiple subsystems doing many different jobs. The simulation class will be the main frame. CommandPallete class is where it includes methods such as setting the simulations map, start simulation, stop simulation, and quit simulation. The map class will include functions that will add, get, and set the cars. This system will handle accidents with accident_handlr and other set the approratte path using the find_path method. Another function is also included in the map called place tiles which will help set the objects of the map. Finding a path is also a function that will be used to find the appropriate path for the vehicles. And lastly for the map class a accident handler function is provided to handle accidents. The car class includes functions such as moving, velocity, and setting the state of the car. Tiles is another class that is abstract and includes setting the coordinates

needed for the tiles. The tiles class includes multiple subclasses such as road, building, stop sign, intersection, and traffic lights. These are all subclasses of tiles and will be presented in the simulation when ran. The Map and Tiles class are aggregated with the simulation class through interface. Finally, there is a coordinate class that calculates/sets coordinates and manhattan distance.



- We chose the model view architecture because of several reasons. For example, the model shows the data from CSV and associates itself with the map. Data is being used. The view helps interact with the user and its interface which is what we are doing. The user is interfering with functions and classes to create a structured view. The controller would take the input for example from place tiles and run and stop. With all the information is has it will use it the control the simulation and portray what it will do. MVP is a great architecture for our project.

Hardware/software mapping

No hardware implementation done in this simulator.

Persistent data management

Different model of cities, maps, cars, tiles,.. will be stored inside CSV file(s).

Access control and security

For our control and security we decided to have a login where the user can login before starting the simulation. This will help protect the simulation that the mayor will be using and testing on.

Global software control

The system will be ran through gui. We will use python to code and start the mapping and controls of system through tkinter that communicates with the interface. Requests are given by the user and the user has the ability to start, stop, or reset/quit the system. When the user starts the simulation the CommandPallete class will operate the method start_button() to start the simulation. When stopping the simulation, the CommandPallete class will use the pause_button() method where it will pause the game exactly where it is. When ever the user wants to reset the simulation per request it will use the quit() method that will quit the simulation and get back to its start phase. The CommandPallete interacts with the user interface and is associated with the map class where the simulation occurs. Map is than aggregated with the simulation, so is the Tiles class. The Tiles class include multiple child classes that will interact with the Tiles class when used. The child classes are Car, Road, Building, Stop Sign, Intersection, Traffic Lights. These subsystems will help provide the needed objects when communicating with Tiles because the Tiles class will use these subsystems, that it interacts with to portray the objects appropriately. Tiles will switch between the objects.

Boundary conditions

-The start-up of our design includes the start of the simulation itself. Vehicles will start entering the map while its count is being calculated. The shutdown simply stops the simulation exactly where its current state. Simple error behaviors that occur in the system are accounted for. For example, when accidents occur the system will take care of this by simply resetting the system.

Subsystem Services

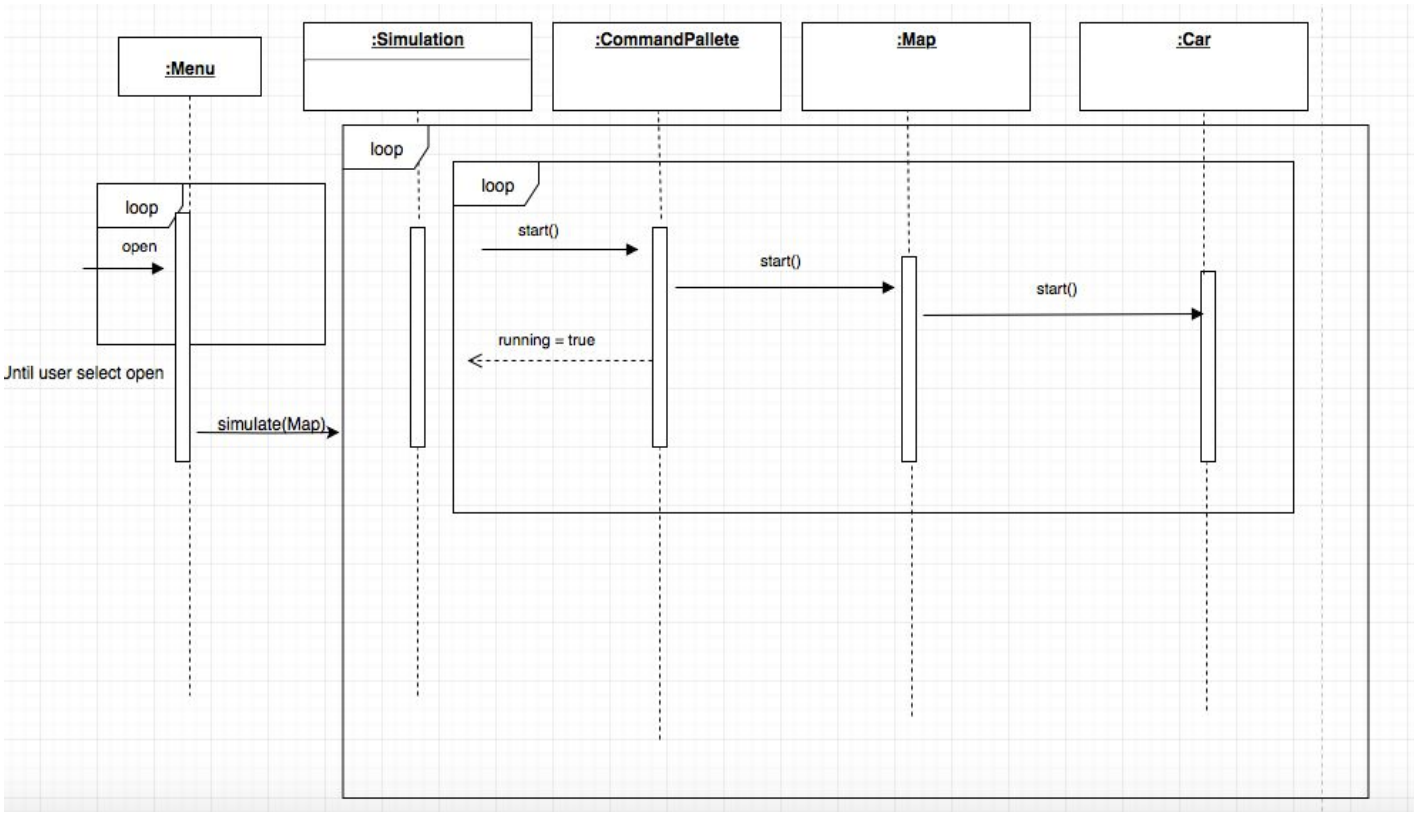
Subsystems such as map, tiles, and coordinate of the simulation will help set up the maps, its coordinates and the tiles for a nice frame and architectural look. The other subsystems such as stop sign, traffic lights, roads, buildings, and intersections also will provide the necessary objects needed for the other subsystems in order for the simulation to properly function. First of all, the CommanPallete subsystem will properly calculate the time when the simulation starts with the start_timing method. The start_button method will help by starting the simulation. pause_button will stop the simulation exactly where it is. The quit() method will then shut the system and log out of it. The Map subsystem will use methods such as add_car, get_car, and set_car. These methods will help add get and set the car appropriately. The find_path method will help find the most successful path set by the system. The accident_handlr method will handle any scenario where accidents might occur. The Tiles subsystem will help set the appropriate objects. Other classes are connected with the tiles subsystem such as Road, Car, Stop sign, Intersection Traffic Lights, Building. These classes are all associated with the tiles class into one whole subsystem. The coordinates class is also part of the full subsystem of the tiles class. The Coordinates class will help set the right coordinates and calculate the manhattan distance of the vehicles. Coordinates will be included in the subsystem of the tiles classs. These objects will play a big role when creating the map. Tile subsystem is associated with the Map subsystem which will set the appropriate objects and set the architecture of the system. Map and Tile subsystems are aggregated with the simulation subsystem. The menu class is a subsytem with the similation class. When the menu is open the simulation is not used until user decides to.

Class Interfaces

Our class diagram consist of multiple classes that interact with each other in order to successfully function and do their jobs. The simulation class is aggregated with other subsystems that help provide appropriate features. For example, the map and tiles classes are all aggregated towards the simulation class. These interactions will help create and base the necessary features needed. The map class includes methods such as add_car, get_car, and set_car. These methods will appropriately deal with the cars in the map. Other methods such as place_tiles, find_path, and accident_handlr are included. find_path will help find the most efficient path. accident_handlr will handle situations of accidents. The map class is associated with the tiles class through the method place_tiles which will place the neccessary tiles. The tiles class includes getting its type and it is associated with the coordinate class. This will help create the coordinates needed to set the tiles for the user to appropriately use. The coordinate class is also associated with the tiles class where it includes methods such as mahattan_distance where is finds neccessary distance. The tiles class are associated with subclasses such as car, road, building, stop sign, intersection, and traffic light. The cars class includes methods such as move which shows its movement. The update_state, update_velcocity methods help show the velocity and state of the vehicle on the map and when it near tiles. The rest of the subclasses include its own set_type methods which sets their types and features that will be displayed in the map. Traffic Light class will have its own methods called update which will update the traffic lights with the colors red, orange, and green at appropriate times. The commandpallete class is associated with the map class where the methods in commandpallate will be invoked. Methods such as start timing, start button, stop button, quite, and step. The start timing will calculate the time the simulation is being ran and when it stops. Start button will start the whole simulation and the stop button will stop the simulation where ever it is at. The quit method will kill the whole simulation and exit out. The step method is a form of resetting the simulation itself. The menu class is a class aggregated with the simulation class. The menu class just like the simulation class works with t kinter with this it will have its own option and if a specific option is selected than the simulation opened.

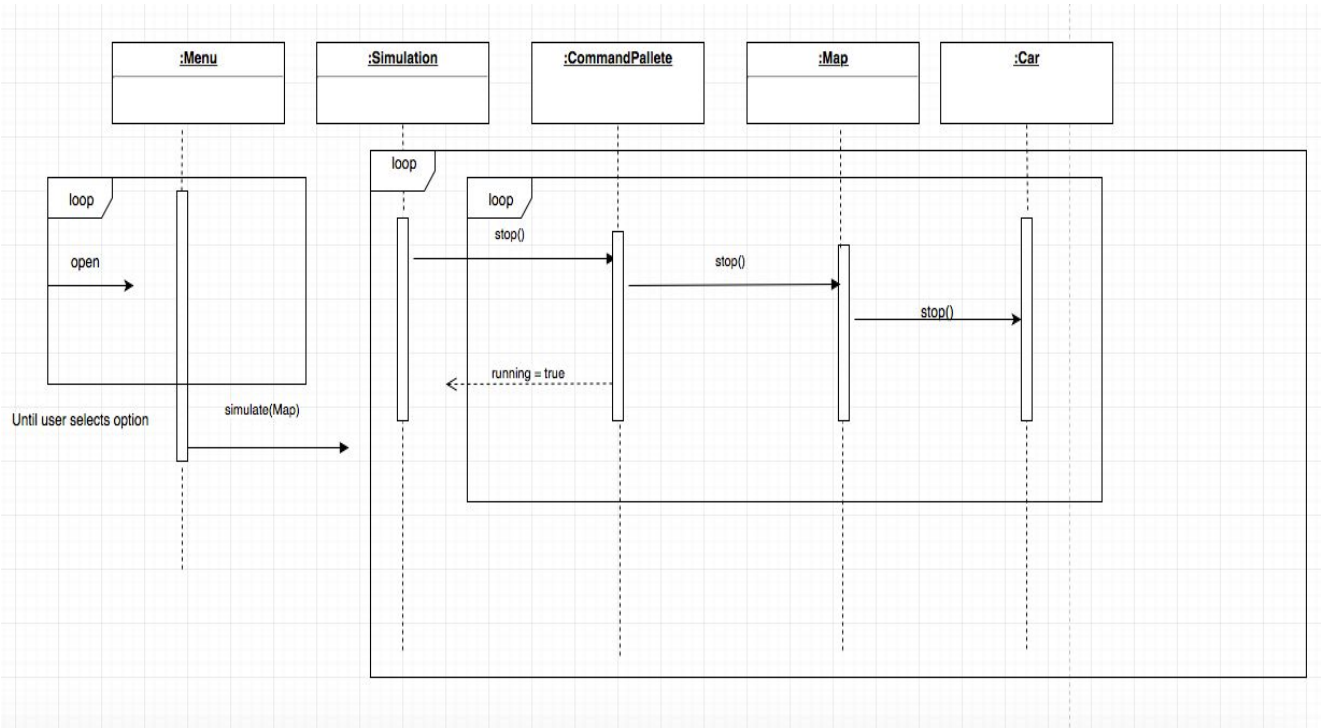
Detailed Design

START SIMULATION SEQUENCE DIAGRAM:



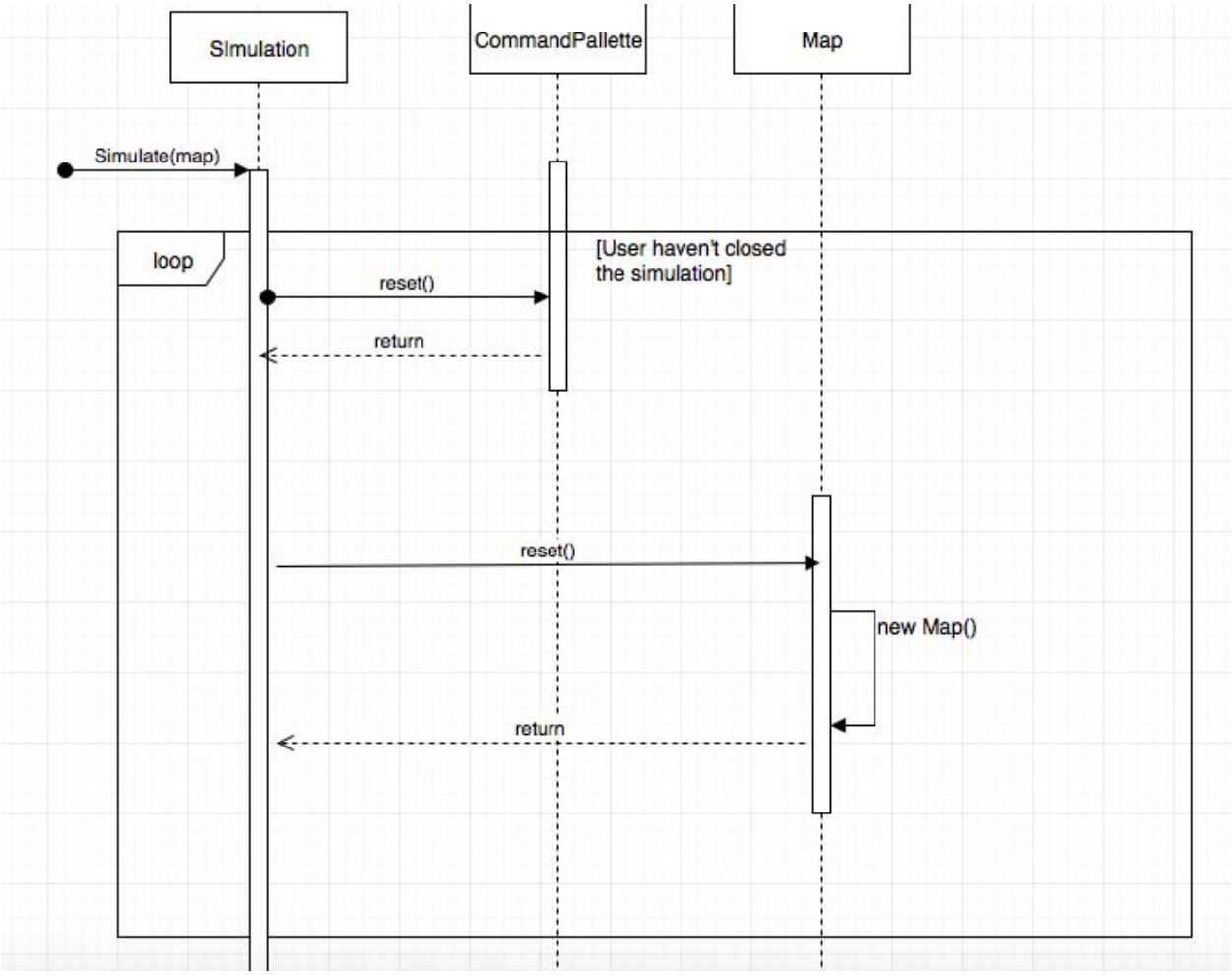
Here we have the open pointing to the menu which is the feature of the menu. The open comes from tk, this will be in loop. The simulation (map) is also connected with the me through the simulation class. The simulation(map) comes from simulation and it sets up the map. The start() come from the command palette and where the start button is invoked and used by user. Returns a boolean true to the simulation. Command palette association with the map allows it to invoke the start button. The cars will be affected by this and the car object will do its job. All of this is within a loop within a loop except for menu.

STOP SIMULATION SEQUENCE DIAGRAM:



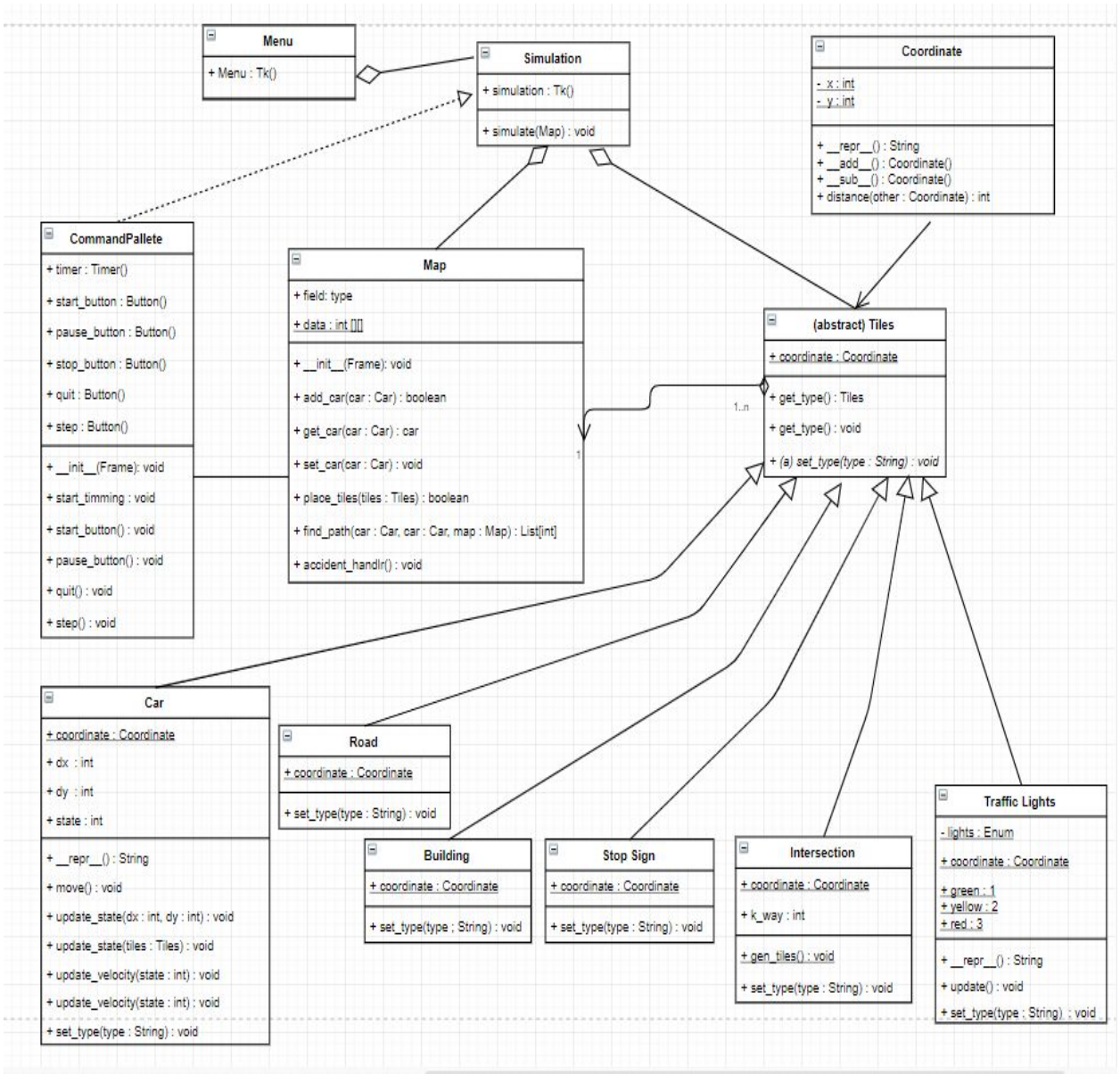
Here we have the open pointing to the menu which is the feature of the menu. The open comes from tk. The simulation (map) is also connected with the me through the simulation class. The simulation(map) comes from simulation and it sets up the map. The stop() come from the command palette and where the stop button is invoked and used by user. Returns a boolean true to the simulation. Command palette association with the map allows it to invoke the stop button. The cars will be affected by this and the car object will do its job. All of this is within a loop.

RESET SIMULATION SEQUENCE DIAGRAM:



- The simulation(map) comes in play from the simulation class. The reset() method from command pallete is used to help invoke reseting the simulation. and it will return. The reset() method also comes in play when dealing with the map because it will reset the map itself. The map will call itself which will than reset the map entirely (Map()). And it is finally returing again to the simulation. All of this is within a loop.

CLASS DIAGRAM:


Menu
+ Menu : Tk()
Simulation
+ simulation : Tk()
+ simulate(Map) : void
Coordinate
- x : int
- y : int
+ __repr__() : String
+ __add__() : Coordinate()
+ __sub__() : Coordinate()
+ distance(other : Coordinate) : int
CommandPallete
+ timer : Timer()
+ start_button : Button()
+ pause_button : Button()
+ stop_button : Button()
+ quit : Button()
+ step : Button()
+ __init__(Frame): void
+ start_timming : void
+ start_button() : void
+ pause_button() : void
+ quit() : void
+ step() : void
Map
+ field: type
+ data : int [][]
+ __init__(Frame): void
+ add_car(car : Car) : boolean
+ get_car(car : Car) : car
+ set_car(car : Car) : void
+ place_tiles(tiles : Tiles) : boolean
+ find_path(car : Car, car : Car, map : Map) : List[int]
+ accident_handlr() : void
(abstract) Tiles
+ coordinate : Coordinate
+ get_type() : Tiles
+ get_type() : void
+ (a) set_type(type : String) : void
1..n
1
Car
+ coordinate : Coordinate
+ dx : int
+ dy : int
+ state : int
+ __repr__() : String
+ move() : void
+ update_state(dx : int, dy : int) : void
+ update_state(tiles : Tiles) : void
+ update_velocity(state : int) : void
+ update_velocity(state : int) : void
+ set_type(type : String) : void
Road
+ coordinate : Coordinate
+ set_type(type : String) : void
Building
+ coordinate : Coordinate
+ set_type(type : String) : void
Stop Sign
+ coordinate : Coordinate
+ set_type(type : String) : void
Intersection
+ coordinate : Coordinate
+ k_way : int
+ gen_tiles() : void
+ set_type(type : String) : void
Traffic Lights
- lights : Enum
+ coordinate : Coordinate
+ green : 1
+ yellow : 2
+ red : 3
+ __repr__() : String
+ update() : void
+ set_type(type : String) : void

GLOSSARY

OOP : Object-oriented programming - An activity concerned with modeling the solution domain with objects.

GUI : Graphical user interface - a user interface that includes graphical elements such as windows, icons and buttons.

UML : Unified Modelling Language - a general-purpose, developmental, modeling language in the field of software engineering.

Python - a high-level general-purpose programming language.

Django : Backend python platform for web development - High-level Python Web framework that encourages rapid development and clean design.

HTML : Hypertext Markup Language - a standardized system for tagging text files to achieve font, color, graphic, and hyperlink effects on World Wide Web pages.

XML : Extensible Markup Language - a metalanguage which allows user to define their own customized markup languages.