Frankfurt University of Applied Sciences

– Faculty of Computer Science and Engineering –

Real-Time Traffic Simulation with Java
Final Report

Project Report for the course

Object-Oriented Programming with Java

submitted on January 18, 2026 by

1655931 Bui, Huy Hoang
1647833 Dao, Gia Hung
1651339 Dang, Huu Trung Son
1651313 Pham, Khac Uy
1387463 Shah, Raees Ashraf

Professor : Prof. Ghadi Mahmoudi

Declaration

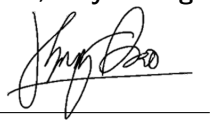
I hereby declare that the submitted project is my own unaided work or the unaided work of our team. All direct or indirect sources used are acknowledged as references.

I am aware that the project in digital form can be examined for the use of unauthorized aid and in order to determine whether the project as a whole or parts incorporated in it may be deemed as plagiarism. For the comparison of my work with existing sources I agree that it shall be entered in a database where it shall also remain after examination, to enable comparison with future projects submitted. Further rights of reproduction and usage, however, are not granted here.

This work was not previously presented to another examination board and has not been published.

Frankfurt am Main, January 18, 2026

1655931 Bui, Huy Hoang



1647833 Dao, Gia Hung

1651339 Dang, Huu Trung Son

1651313 Pham, Khac Uy

1387463 Shah, Raees Ashraf

Contents

Abstract **1**

1 Project Overview **2**

1.1 Introduction . . . 2
1.2 Background . . . 2
1.3 Objective . . . 2
1.4 Technology Stack . . . 3

2 Project Workplan **4**

2.1 Timeline . . . 4
2.1.1 Final Milestone Status . . . 4
2.1.2 Timeline for Final Application . . . 4
2.2 Task Distribution . . . 5
2.3 Challenges and Risks . . . 5
2.4 Project Progress Summary . . . 6
2.5 Software Engineering Practices (Git) . . . 6

3 Application Features **7**

3.1 Graphical User Interface (GUI) . . . 7
3.2 Core Functionalities . . . 8
3.2.1 Live SUMO Connection and Simulation Control . . . 8
3.2.2 SUMO Traffic Network File Organization . . . 9
3.2.3 Map Visualization in the Application . . . 10
3.2.4 Vehicle Injection . . . 10
3.2.5 Traffic Light Control . . . 11
3.2.6 Hotkey Functionality . . . 11
3.2.7 Filtering and Diagnostics . . . 12
3.3 Technical Implementation . . . 12
3.3.1 Simulation Loop Architecture . . . 13
3.3.2 Efficient Vehicle State Management . . . 13
3.3.3 Logger and Run Scripts . . . 14

3.3.4 CSV and PDF Export . . . 14
3.3.5 Map Rendering Engine . . . 15
3.3.6 Concurrency and Thread Safety . . . 16
3.3.7 Data Flow and System Architecture . . . 16
3.4 Class Diagram Overview . . . 17
3.5 Use Case Diagram . . . 20
3.6 Code Documentation and User Guide . . . 20
3.7 Stress Test Scenario and Results . . . 21
3.8 Traffic Network Preparation . . . 21
3.9 SUMO Network and Initial Simulation Test . . . 22
3.10 TraCI Connection Test . . . 23

Conclusion and Future Work **24**

3.11 Conclusion . . . 24
3.11.1 Performance Evaluation . . . 24
3.11.2 Advanced Traffic Control . . . 24
3.11.3 Team Reflection . . . 25
3.11.4 Reporting and Analytics . . . 25

Abstract

This project develops a real-time traffic simulation application using Java, JavaFX, and the Simulation of Urban Mobility (SUMO). The goal is to provide an interactive desktop GUI that connects to a running SUMO simulation via TraCI (TraaS) and enables users to visualize and control key traffic elements.

By the final milestone, the project has been fully implemented with completely functional features with few to no runtime errors. The application can establish a live SUMO connection, advance the simulation in a continuous manner (or step-by-step), and render the selected SUMO network directly inside the JavaFX GUI with moving vehicles. The animation system is fine-tuned to render granular changes in movement, with 20 granular images to render for every time step (step-length of 50 ms). It also supports interactive vehicle injection (selecting a target edge, number of vehicles, speed, and color) and manual traffic light control (switching phases and adjusting phase duration). Furthermore, hotkey functionality has been incorporated to enable swift state changes without the need for cursor interaction. For usability and debugging, the GUI includes live status indicators, a vehicle table and chart, and optional filters (e.g., by vehicle color, speed, and congestion heuristics).

In addition to the working application, the final deliverables include inline code documentation (Javadoc), a complete user guide, and a stress-test scenario (`Stress.sumocfg`) to evaluate performance with a large number of vehicles. The report summarizes implemented features, current limitations, and the main challenges discovered during stress testing. All available resources can be accessed from the project's Github repository. Details are provided below.

Chapter 1

Project Overview

1.1 Introduction

This project aims to build a real-time traffic simulation application using Java as the main programming language and its various libraries, such as JavaFX, TraCI as a Service (TraaS), etc. It also utilizes functionalities of the Simulation of Urban Mobility (SUMO) software package to visualize traffic network and traffic flows. The application is designed to provide some basic mobility control functionalities and allows for user's interactive control, enabling its use for teaching, learning, and researching urban mobility.

The concentration of the project is a wrapper runner application that calls functions and accesses objects from external libraries to simulate real-time traffic, resembling a bird's eye view of a complex apparatus with independent components. This is the essence of object-oriented programming (OOP) design.

Working development and documentation of the project can be found in our GitHub repository:

`https://github.com/hunggiadao/realtime-traffic-simulation-java-oop`

1.2 Background

In more detail, the surface of the application is a GUI that is rendered using JavaFX graphical library, which is a Java library that allows for the creation of interface elements like buttons, dropdown menus, and alert boxes. Such elements send command signals to a SUMO instance to initialize and control a running SUMO traffic network. The communication between the GUI frontend and the SUMO simulator is facilitated by SUMO's TraaS library, which provides functions for retrieving traffic data like vehicle statistics, traffic light states, road data, etc., and functions for commanding the network like cycling traffic lights, spawning vehicles, etc.

When a SUMO instance finishes running, its simulation results are exported in the comma separated values (CSV) format for further inspection and improvement of the application. Our documentation and project report include information about the application's features, usability, architecture diagrams, class design diagrams, and a summary of our development process.

1.3 Objective

The primary objective of this project is to develop a comprehensive real-time traffic simulation application that demonstrates practical implementation of object-oriented programming principles in Java. Specifically, we aim to create an

interactive platform that integrates multiple technologies—JavaFX for the graphical user interface, SUMO for traffic simulation, and TraaS for inter-process communication—into a cohesive system that can be used for educational and research purposes in urban mobility studies.

Beyond the technical implementation, we seek to gain hands-on experience with software engineering practices, including collaborative development using version control systems, systematic documentation of design decisions and development processes, and the creation of modular, maintainable code that follows industry best practices. The project also serves as an opportunity to explore real-world challenges in traffic simulation and control systems.

1.4 Technology Stack

To familiarize ourselves with the necessary programs, we watched various tutorial videos and read documentation on **SUMO**, **Netedit**, and **JavaFX**. We then shared these resources within our team so that everyone could make use of them.

We began our project by taking initial notes and outlining the tasks ahead of us. After that, we decided which tools we would use for collaboration: **Notion** and **Git**. We used Notion to share our Notes and document important information.

At first, we focused on learning how to create and manipulate network files in Netedit. We watched road simulation videos and studied documentation to gradually understand how the program works. We created different simulations, added traffic lights, stop signs, pedestrian paths, and bicycle lanes, and tested their behavior. In the beginning, it was difficult to understand Netedit, but over time we became more confident in using it.

After we became familiar with Netedit and SUMO, we started assigning roles within the team and defining who would be responsible for which tasks. Role and timeline information is provided in the following chapter.

Chapter 2

Project Workplan

2.1 Timeline

2.1.1 Final Milestone Status

The final milestone focused on delivering a fully fletched application with fully functional features. In addition, the application must run without non-reproducible errors. The application also logs all runtime information and saves it into a log file for later inspection. The status is summarized below.

- **Final Application**: Full GUI with map, controls, statistics; Vehicle grouping/filtering; Traffic light adaptation (manual or rule-based); Exportable reports (CSV and/or PDF).
- **Final Documentation**: Updated user guide; Final code documentation; Summary of enhancements and design decisions.
- **Presentation**: Live demo; Architecture and feature explanation o Performance evaluation; Team reflection.
- **Test Scenario:** A stress scenario configuration (`Stress.sumocfg`) is provided to test high vehicle counts.
- **Final Retrospective**
- **Clean Git Repository**
- **Sample Exported Reports**
- **Progress Summary**: Performance and UI-update frequency were identified as key bottlenecks under heavy load.

2.1.2 Timeline for Final Application

- **Full GUI Implementation**: Complete map visualization, controls, and statistics dashboard by 5th January
- **Vehicle Grouping/Filtering**: Implement filtering and categorization features by 5th January
- **Traffic Light Adaptation**: Manual and rule-based control system by 8th January
- **Exportable Reports**: CSV and PDF export functionality by 10th January
- **Final Documentation**: Updated user guide and comprehensive code documentation by 18th January
- **Project Summary**: Enhancements overview and design decisions report by 18th January

- **Presentation**:
  - Live demo 19th January
  - Architecture and feature explanation by 18th January
  - Performance evaluation by 12th January
  - Team reflection by 18th January
- **Final Retrospective** by 18th January
- **Clean Git Repository** by 18th January
- **Sample Exported Reports** by 18th January

2.2 Task Distribution

Team roles for the entire project are as follows:

- **Traffic Network Files**: Raees Ashraf Shah
- **TraCI Connector + SUMO Integration**: Gia Hung Dao
- **Traffic Light Control (wrapper + UI)**: Gia Hung Dao, Raees Ashraf Shah
- **Hotkey Functionality**: Raees Ashraf Shah
- **Vehicle Wrapper + Vehicle Injection**: Huu Trung Son Dang
- **GUI Implementation (JavaFX)**: Huu Trung Son Dang
- **Infrastructure Wrapper Classes**: Huy Hoang Bui, Gia Hung Dao
- **Logger + Run Scripts**: Huu Trung Son Dang
- **Exception handler**: Huu Trung Son Dang
- **CSV and PDF Exporter**: Raees Ashraf Shah, Gia Hung Dao
- **Architecture and UML Diagrams**: Khac Uy Pham, Huu Trung Son Dang

2.3 Challenges and Risks

- **Performance under load**: With many vehicles, frequent UI refreshes (map + table) can become CPU-intensive.
- Some network-wide function calls used to gather entire information take longer than do function calls with specific Object ID.
- **Update frequency trade-offs**: Increasing simulation step length or throttling table/map updates reduces load but lowers UI responsiveness.
- **Robust TraCI lifecycle**: SUMO may terminate or close the socket when a scenario ends; the application must handle disconnects gracefully.
- Some metric functions in `EdgeWrapper` class took longer than other functions, since their runtime complexity grows exponentially with the number of edges and vehicles.

2.4 Project Progress Summary

The final milestone focused on finishing ongoing features from Milestone 2, adding final features, and fixing all runtime bugs. The following summary highlights the status and main outcomes.

- **Status**: All features completed (live SUMO connection, in-app map rendering with moving vehicles, vehicle injection, filtering views, traffic light control, hotkeys, statistics recording, and file exports).
- **Key changes**: integrated vehicle state and edge state retrieval for the table and map, added traffic light phase control, added metrics charts and diagrams, implemented CSV and PDF export features.
- **Stress testing**: Used a dedicated stress configuration (`Stress.sumocfg`) to validate behavior at higher vehicle counts and to observe performance bottlenecks.
- **Challenges**: Under load, UI updates (map redraw + table refresh) can be expensive; practical mitigation is to reduce refresh frequency or increase `step-length`.

2.5 Software Engineering Practices (Git)

- **Branching + Pull Requests**: The main branch is push protected. Development must be performed on feature branches and merged into main via pull requests (e.g., simulation classes, traffic light wrapper updates, UI fixes).
- **Commit messages**: Commits generally use action-oriented prefixes (e.g., *feat*, *fix*, *update*, *add*, *remove*, etc.) and describe the affected subsystem (UI, simulation, wrappers, build).
- **Traceability**: the project work is visibly traceable through incremental commits that introduce the simulator loop and vehicle-management abstractions, followed by bug fixes and UI integration improvements. Commit history can be viewed in the Github repository.

Chapter 3

Application Features

3.1 Graphical User Interface (GUI)

For Milestone 1, we designed the graphical user interface of the application using JavaFX and SceneBuilder. In Milestone 2, we extended this UI into a functional prototype by integrating a live SUMO connection, real-time stepping, map rendering, vehicle injection, and traffic light control. In the final milestone, we added statistical charts and graphs that visualize changing vehicle data in real time.

The interface is built around a `BorderPane` layout, dividing the window into clear functional regions. At the top of the application, a responsive toolbar provides the essential simulation controls, including opening a SUMO configuration file, connecting to the simulation backend, starting or pausing the simulation, executing single simulation steps, and adjusting the simulation speed. This layout remains clean and stable when the window is resized.

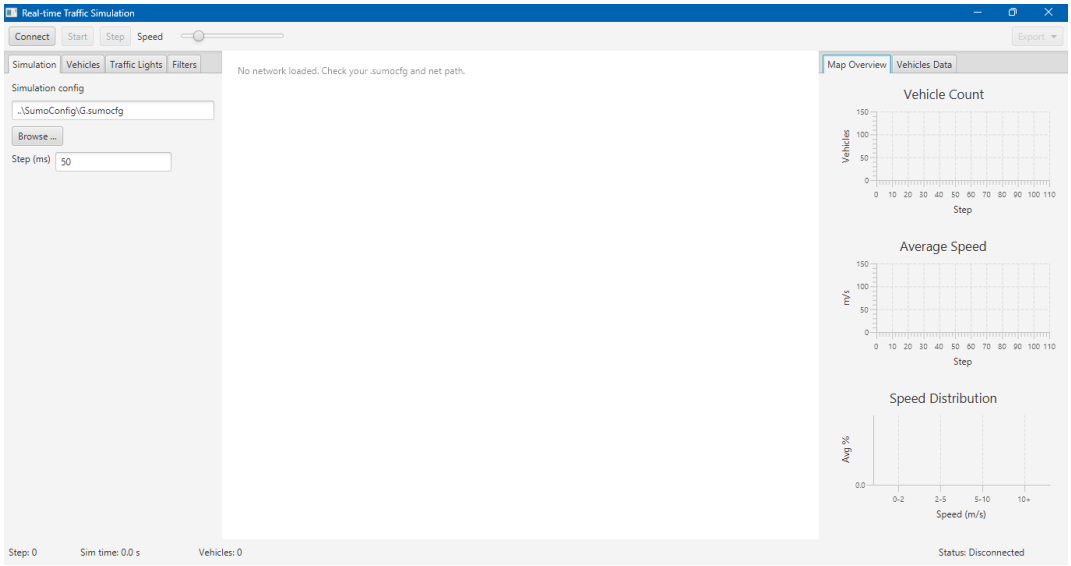


Figure 3.1: Final Graphical User Interface (GUI)

On the left side, a `TabPane` organizes settings into tabs such as Simulation, Vehicles (injection), Traffic Lights, and Filters. The Simulation tab supports selecting a `.sumocfg` file and adjusting the step interval, while the Vehicles tab provides controls for injecting vehicles into a selected edge. The Traffic Lights tab allows the user to view current status

of a specific traffic light and manipulate its state as well, and the Filters tab supports viewing the map under conditional restraints.

The central region contains the map view. In the final milestone, this region renders the SUMO network (parsed from the network file referenced by the loaded `.sumocfg`) and draws moving vehicles, environmental background based on TraCI position updates. The map supports zooming and panning to inspect local behavior around intersections.

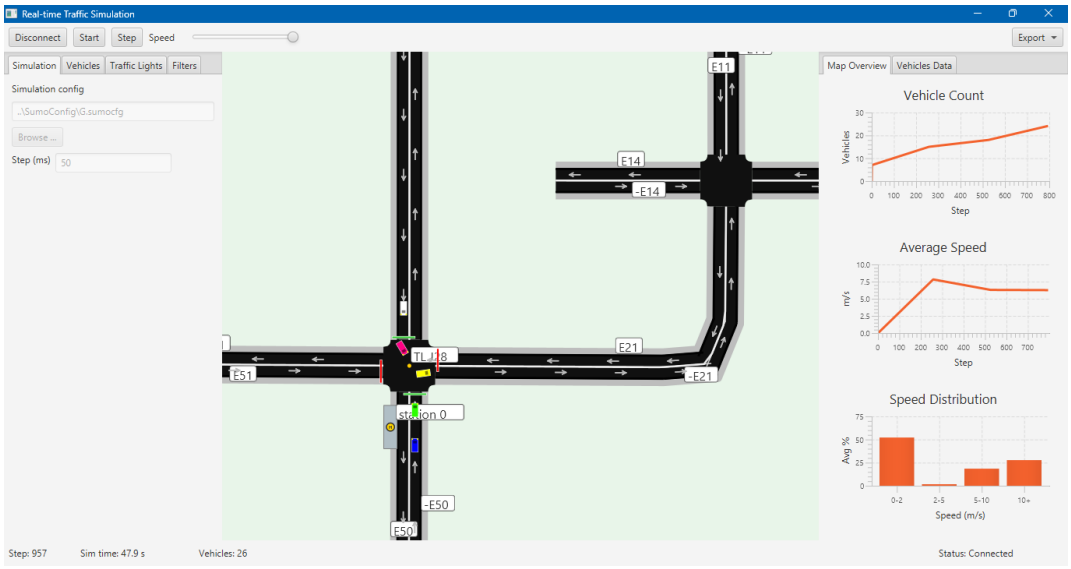


Figure 3.2: Live map rendering and simulation controls in the JavaFX GUI

On the right side, there is a dropdown menu for exporting simulation data as a comma separated values (CSV) file or portable document format (PDF) file. A second `TabPane` provides simulation metrics. In the Map Overview tab, line chart are drawn to track vehicle count, average speed, and speed distribution over time. The Vehicle Data tab contains a JavaFX `TableView` that is populated with live data (vehicle ID, speed, current edge, and color). Detailed vehicle data is organized in a table and a pie chart of their color distribution is also shown.

Finally, a status bar at the bottom displays key runtime information such as simulation step, simulation time, vehicle count, and connection status.

Figure 3.3: Status Bar of the application window

Overall, the UI now provides a complete workflow for the application: load a scenario, connect to SUMO, run or step the simulation, visualize the network with moving vehicles, inject vehicles interactively, manipulate traffic light phases, filter vehicles by different criteria, inspect detailed metrics, and exporting simulation data to external files.

3.2 Core Functionalities

3.2.1 Live SUMO Connection and Simulation Control

The application establishes a TraCI connection to SUMO and manages the simulation lifecycle (connect, disconnect, step, and run/pause). The simulation can run continuously with a configurable step length (default is 50 ms), and the GUI

reflects the current state through a status bar (step number, simulation time, and vehicle count). The UI also remembers the last opened `SumoConfig` file for express running, in case there is no need to change simulation network.

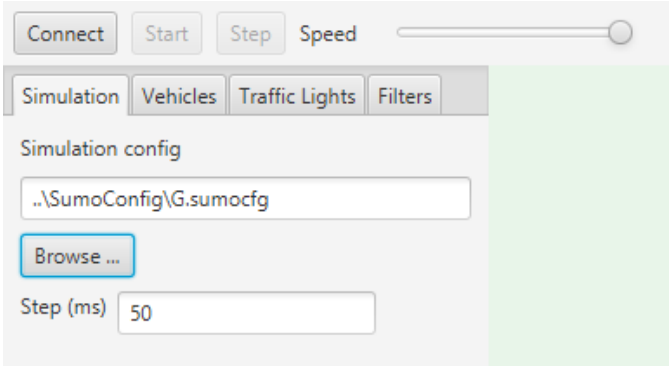


Figure 3.4: TraCI Connection Selection Menu

Additionally, users can press P on the keyboard to toggle Play/Pause for the simulation.

3.2.2 SUMO Traffic Network File Organization

There are 5 types of files associated with each SUMO project. The `.netecfg` file keeps record of all other files in the project, mainly used for the `netedit` GUI, not necessarily for running SUMO simulations. It references the `.sumocfg` file, which is the main storage of all data for that specific simulation. The `.sumocfg` file references 3 other files:

- `.net.xml`: network data, containing data about the map, such as edges, lanes, junctions, traffic lights
- `.rou.xml`: route data, containing data about routes, traffic flows, vehicles, etc.
- `.add.xml`: additional data, containing data about bus stops, charging stations, parking areas

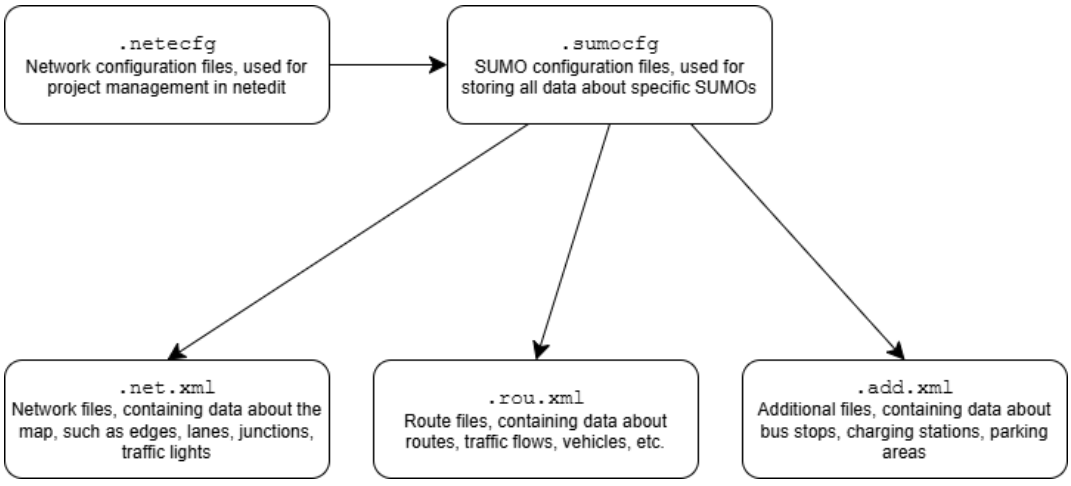


Figure 3.5: SUMO Traffic Network File Structure

3.2.3 Map Visualization in the Application

The map renderer parses the SUMO network file (`.net.xml`) to extract lane polylines and junction shapes. During runtime, the renderer updates vehicle positions from TraCI and draws moving vehicles on top of the network. A key feature introduced in Milestone 2 is the interactive zoom and pan capability, allowing users to inspect specific intersections or view the entire network at a glance. To improve interpretability at intersections, the map overlays traffic-light stop lines colored by the current signal state (red/yellow/green), providing immediate visual feedback on signal phases.

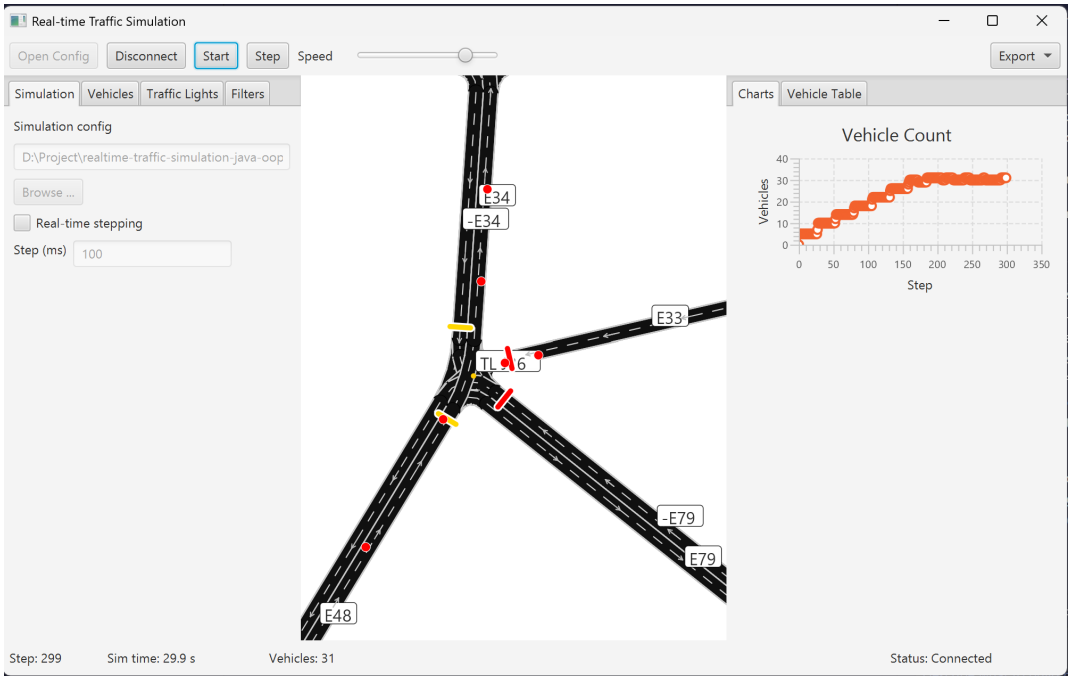


Figure 3.6: Zoomed-in map view with moving vehicles and traffic-light stop-line overlays (red/yellow/green).

3.2.4 Vehicle Injection

Users can inject vehicles during a running simulation by selecting a target edge and specifying the number of vehicles, an optional speed, and a vehicle color. The ability to customize vehicle color was recently added to help visually distinguish injected fleets from background traffic. Injected vehicles are immediately reflected in the map rendering and vehicle table, enabling quick scenario exploration and load testing.

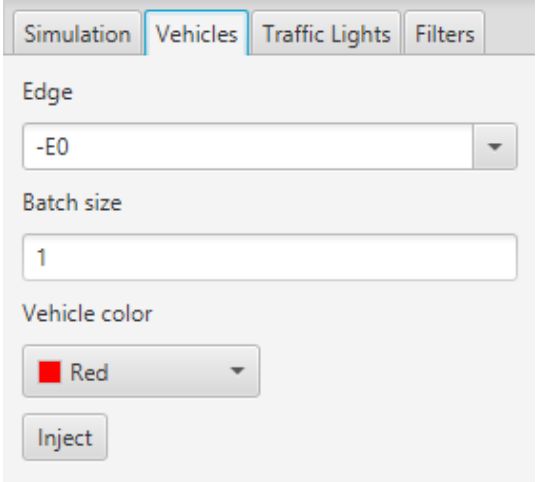


Figure 3.7: Vehicle injection controls used during stress testing, including color selection.

3.2.5 Traffic Light Control

The Traffic Lights tab provides direct control over signal behavior. Users can select a specific traffic light ID from the network. The interface then displays the current phase and state definition. Operators can manually switch to the previous or next phase and dynamically adjust the current phase duration. This enables interactive experimentation with signal timings and their impact on traffic flow, which is visualized in real-time on the map.

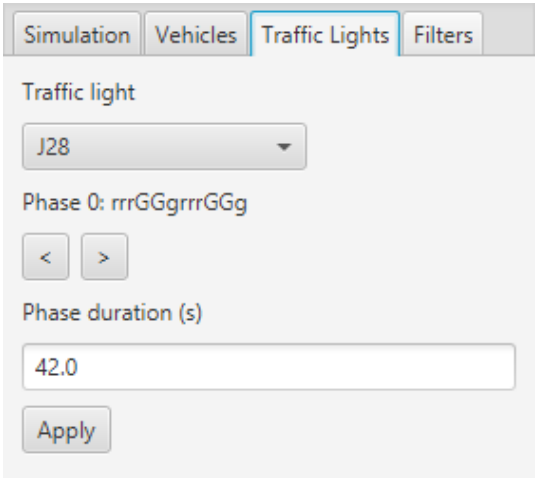


Figure 3.8: Traffic light selection and manual control: phase/state display and phase duration adjustment.

3.2.6 Hotkey Functionality

Hotkey functionality has been added for the final milestone. There are several keyboard inputs users can press to initiate certain actions quickly, without having to use their cursor or switch to a different UI tab. Users can press:

- **LEFT** arrow key to cycle to the previous traffic light
- **RIGHT** arrow key to cycle to the next traffic light

- **UP** arrow key to immediately transition the current traffic light to the next phase
- **DOWN** arrow key to transition the current traffic light to the previous phase
- **P** to toggle Play/Pause for the simulation

3.2.7 Filtering and Diagnostics

To support debugging and analysis, the GUI provides optional vehicle filters based on (1) vehicle color, (2) speed threshold, and (3) a congestion heuristic (edge mean speed and/or vehicle speed). The vehicle table displays the filtered set, and a chart plots vehicle count over time.

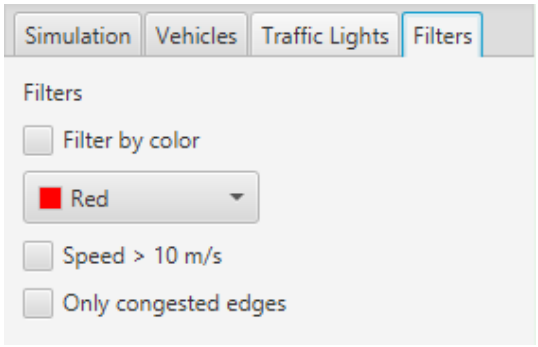


Figure 3.9: Filter controls used to isolate vehicles by color, speed, and congestion heuristics

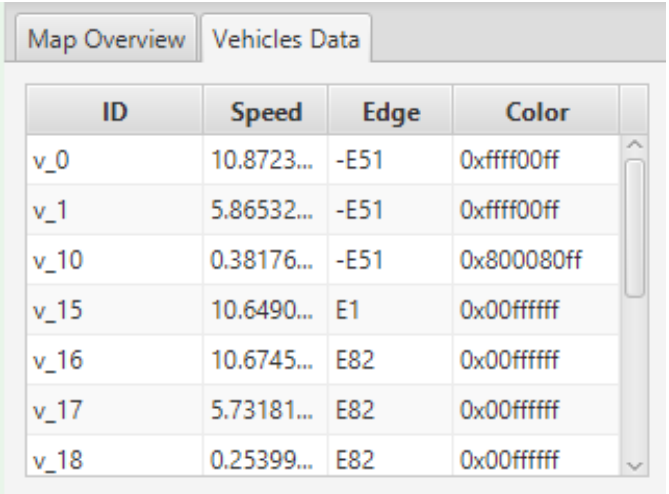


Figure 3.10: Vehicle table populated with live simulation data (filtered view option available)

3.3 Technical Implementation

To support the functional requirements of multi-threading and concurrency, specifically the need to handle high vehicle counts and real-time visualization, we implemented a robust architecture separating the simulation logic from the UI rendering. This separation is further discussed in the Concurrency and Thread Safety section to explain how the UI can function independently from the underlying simulation runtime.

3.3.1 Simulation Loop Architecture

The core simulation loop is encapsulated in the `VehicleSimulator` class. To ensure responsiveness and prevent the user interface from freezing during heavy computation, the simulation runs on a dedicated single-threaded executor. This design decouples the TraCI communication—which is synchronous and blocking—from the JavaFX Application Thread.

The simulator uses an atomic state management approach. At each step, it triggers the `VehicleManager` to refresh the list of active vehicles and update their states. This ensures that the simulation state is consistent before being passed to the UI for rendering.

3.3.2 Efficient Vehicle State Management

Handling thousands of vehicles requires minimizing the overhead of TraCI calls. The `VehicleManager` and `VehicleWrapper` classes implement a caching strategy where vehicle data (position, speed, color) is fetched in bulk or lazily updated.

We introduced the `VehicleState` class, an immutable data structure that captures a snapshot of a vehicle at a specific simulation step. This immutability allows the simulation thread to safely pass data to the UI thread without complex synchronization locks, avoiding race conditions where the UI might try to render a vehicle that is being modified.

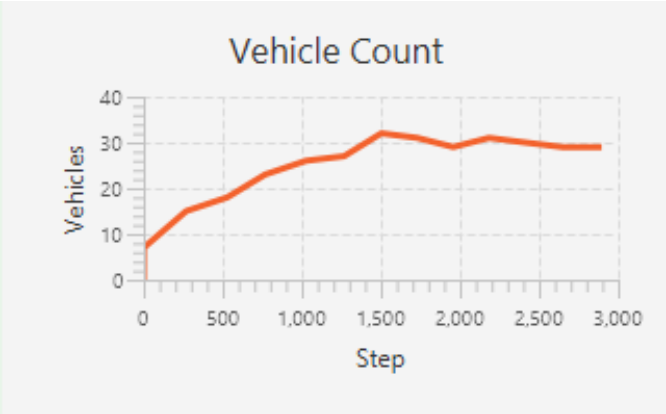


Figure 3.11: Vehicle Count Chart: the number of vehicles over time steps

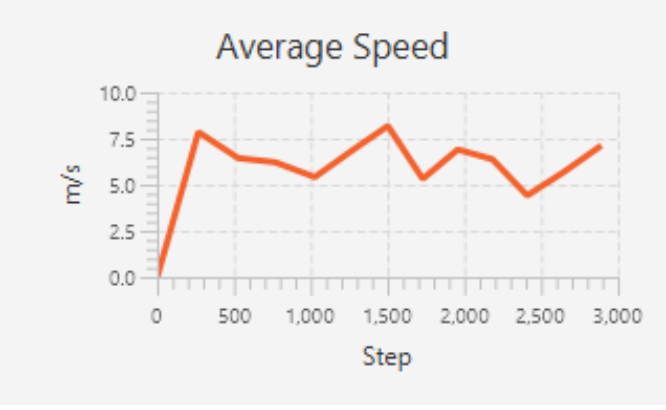


Figure 3.12: Average Speed Chart: Average speed of all vehicles over time steps

Figure 3.13: Speed Distribution Chart: the distribution of speeds of vehicles across multiple ranges

3.3.3 Logger and Run Scripts

To smoothen the running process, we have created a `run.bat` batch file for running automation. The user can simple double-click the file in a GUI window, or invoke its execution in a command line interface (CLI) environment to start the application.

A `Logger` class is also included in the program to log various events and results of functions during the simulation runtime. This helps with debugging and monitoring the running state of the application. The logger also reports any error whenever an exception occurs during execution. It prints the logged message out to the terminal and saves all the records in a standalone app.log file for future inspection. Running logs are saved in the `/logs` folder of the project's root directory.

3.3.4 CSV and PDF Export

By the final milestone, an `Export` class has been implemented to export snapshots of simulation metrics and statistics (vehicle IDs, colors, speeds, positions, edge occupation, traffic light IDs, state, phase index, etc.) for post simulation summary. The export data can exist in comma separated values (CSV) form or portable document format (PDF) form. The PDF export contains more customized data filtering, such as tables of vehicles by speed and color, and additional system information to give users a better overview of the simulation.

By default, the exporter depends on the immutability of states of values in the simulation, so the menu is only accessible when the simulation is paused. When the simulation is in `Running` state, the menu is disabled.

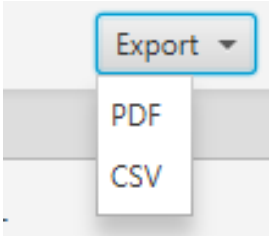


Figure 3.14: Export dropdown menu: allows for PDF and CSV exporting

```
Step,Vehicle-ID,Color [R-G-B-A],Speed [m/s],PosX,PosY,Edge,TrafficLight-ID,Current State,Phase Index
0,f_0.21,255-0-133-255,12.42,159.32,-47.22,E5,J0,rrGGGr,2
1,f_0.22,255-0-133-255,12.94,-99.65,-107.61,E22,J28,rrrGGgrrrGGg,0
2,f_1.20,255-127-0-255,14.16,-40.49,164.20,E24,J37,GGgGGrrrrrrGgGg,0
3,f_1.21,255-127-0-255,9.39,103.81,44.89,E3,J40,GGGG,0
4,f_1.22,255-127-0-255,0.00,299.50,290.13,E49,J50,yyyy,1
5,f_3.21,0-255-231-225,11.40,95.32,302.03,E47,J52,rrrrrGGgG,2
6,f_3.22,0-255-231-225,3.77,162.42,80.85,E4,J54,rrrrrrryyyy,3
7,f_4.43,240-0-255-255,11.00,140.03,23.62,E2,,,
8,f_4.44,240-0-255-255,13.18,-101.01,50.69,E36,,,
9,f_5.43,255-255-255-255,0.00,-99.71,-236.61,E22,,,
10,f_5.44,255-255-255-255,9.87,-27.72,26.40,-E6,,,
11,f_6.33,28-228-35-255,0.00,-99.71,-244.11,E22,,,
12,f_6.34,28-228-35-255,7.03,-99.70,-216.97,E22,,,
13,f_6.35,28-228-35-255,13.93,102.11,113.62,-E3,,,
14,f_7.34,192-192-192-255,11.86,100.96,26.79,:J2_4,,,
15,f_7.35,192-192-192-255,0.00,292.02,290.69,E49,,,
16,f_8.30,255-63-78-255,15.31,24.93,224.65,E28,,,
17,f_8.31,255-63-78-255,11.56,-94.44,23.94,:J7_8,,,
```

Figure 3.15: CSV export sample

Statistics and Metrics

Timestamp	2026-01-17 02:51:52
Total edges	226
Total lanes	307
Total traffic lights	7
Total vehicles	18
Total congested vehicles	4
Total wait time	94.05000000000001
Average vehicles per edge	0.07964601769911504
Fastest vehicle	f_8.30
Fastest vehicle speed	15.307576722633435
Slowest vehicle	f_1.22
Slowest vehicle speed	0.0
Average vehicle speed	8.768315580793807

All Traffic Light Data

Step	TrafficLight-ID	Current State	Current Phase Index
0	J0	rrGGGr	2
1	J28	rrrGGgrrrGGg	0
2	J37	GGgGGrrrrrrGgGg	0
3	J40	GGGG	0
4	J50	yyyy	1
5	J52	rrrrrGGgG	2
6	J54	rrrrrrryyyy	3

Figure 3.16: PDF export sample

3.3.5 Map Rendering Engine

The `MapView` component is a custom JavaFX `Canvas` implementation designed for performance. Instead of using heavy scene graph nodes for each map element, it performs direct drawing operations.

Network Parsing: On initialization, the renderer parses the SUMO network file (`.net.xml`) using a DOM parser to extract lane geometries and junction shapes. These are stored as lightweight `LaneShape` and `JunctionShape` objects.

Coordinate Transformation: The renderer implements a dynamic coordinate transformation system that maps SUMO's Cartesian coordinates to the screen space, supporting real-time zooming and panning.

Layered Rendering: The draw loop is optimized to render static elements (roads, junctions) first, followed by dynamic overlays (traffic light states, vehicles). This ensures that dynamic elements and critical information always stay visible on top.

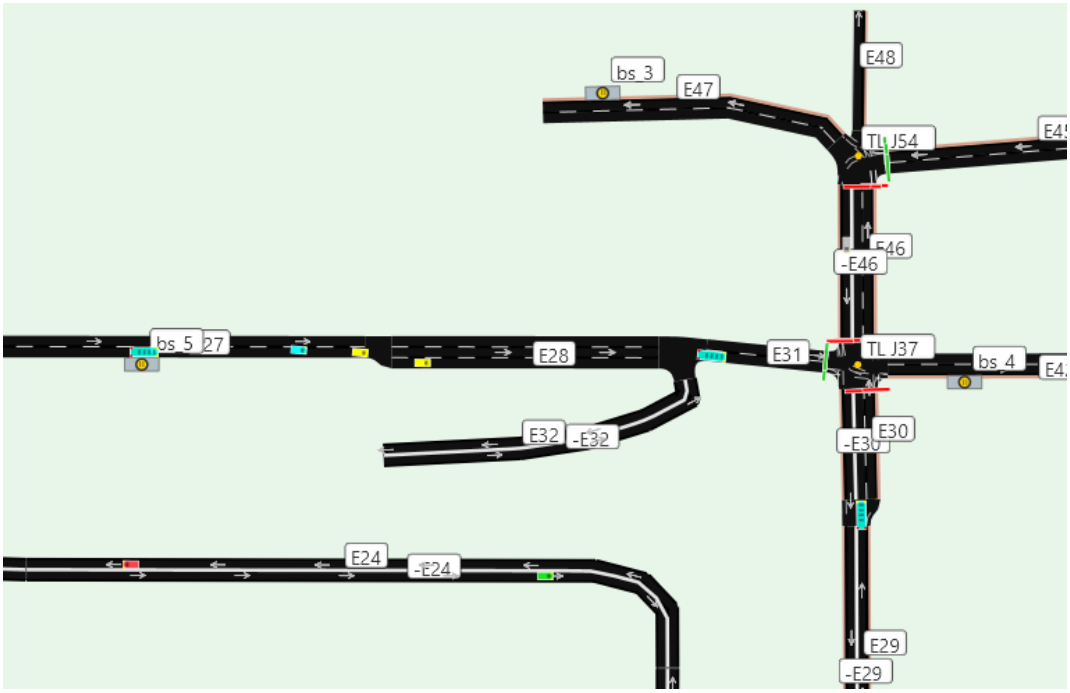


Figure 3.17: Map rendering engine: layered rendering

3.3.6 Concurrency and Thread Safety

A key challenge in Milestone 2 was bridging the gap between the synchronous TraCI simulation and the asynchronous JavaFX UI. We solved this using a producer-consumer pattern:

- The **Simulation Thread** (Producer) steps the simulation and produces a map of `VehicleState` objects.
- The **UI Thread** (Consumer) polls this state via a thread-safe reference and updates the `MapView` and `TableView`.

This architecture allows the simulation to run at maximum speed (or a fixed real-time step) without being throttled by the frame rate of the UI, while the UI renders the latest available state as smoothly as possible.

3.3.7 Data Flow and System Architecture

The data flow within the application is designed to be unidirectional where possible to maintain state consistency.

1. **Input**: The user interacts with the GUI (e.g., clicking **Inject Vehicle**).
2. **Command**: The GUI sends a command to the `VehicleManager` or `TrafficLightWrapper` classes.
3. **Simulation**: The `VehicleSimulator` executes the command via TraCI during the next simulation step.
4. **Update**: SUMO computes the new state.
5. **Fetch**: The `VehicleManager` fetches the updated vehicle positions and statuses.
6. **Render**: The `MapView` receives the immutable `VehicleState` snapshot and redraws the scene.

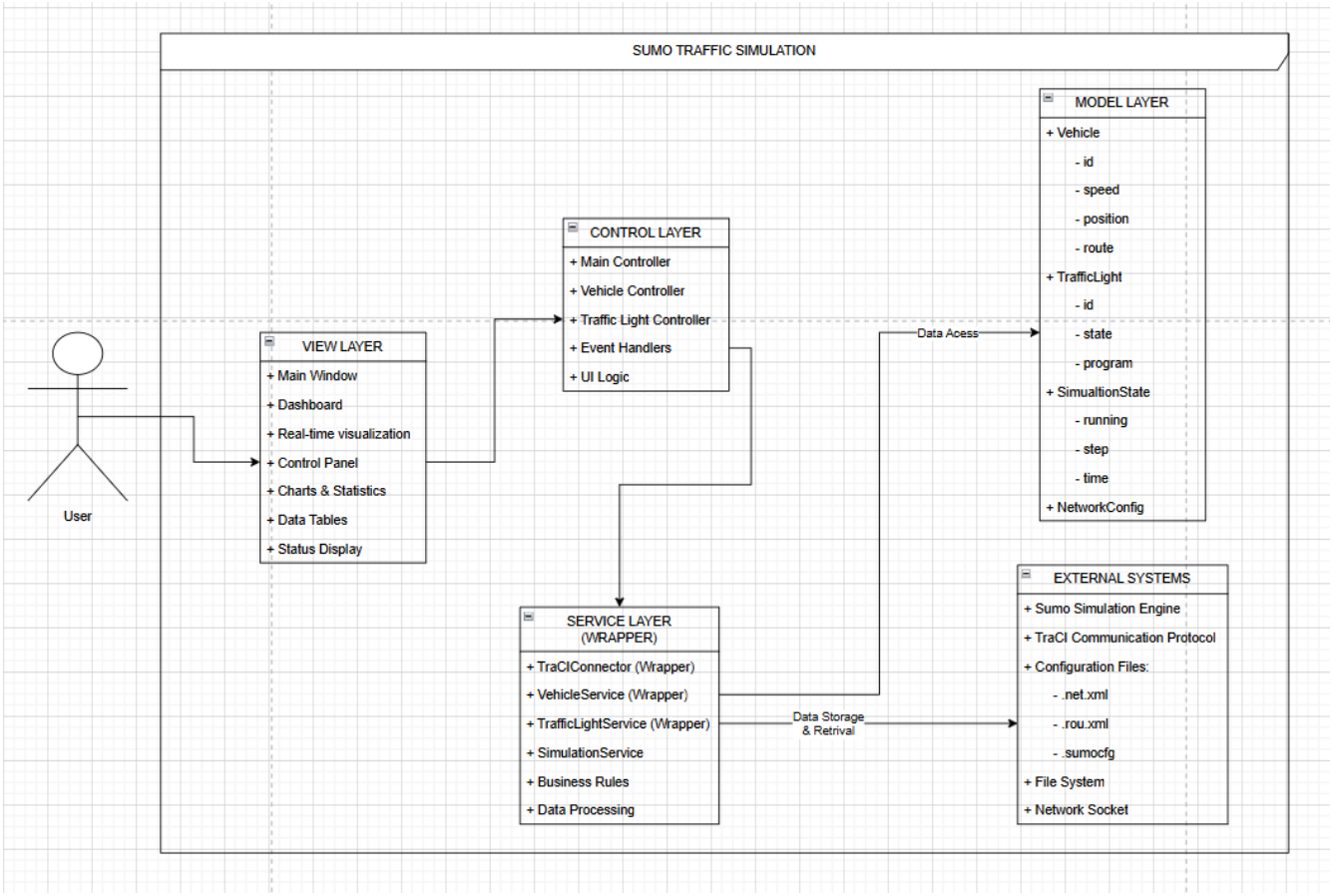


Figure 3.18: System Architecture Diagram illustrating the layered design and data flow between the JavaFX GUI, Controller, Wrapper, and SUMO.

3.4 Class Diagram Overview

Figure 3.19 shows the simplified class structure implemented for the final milestone.

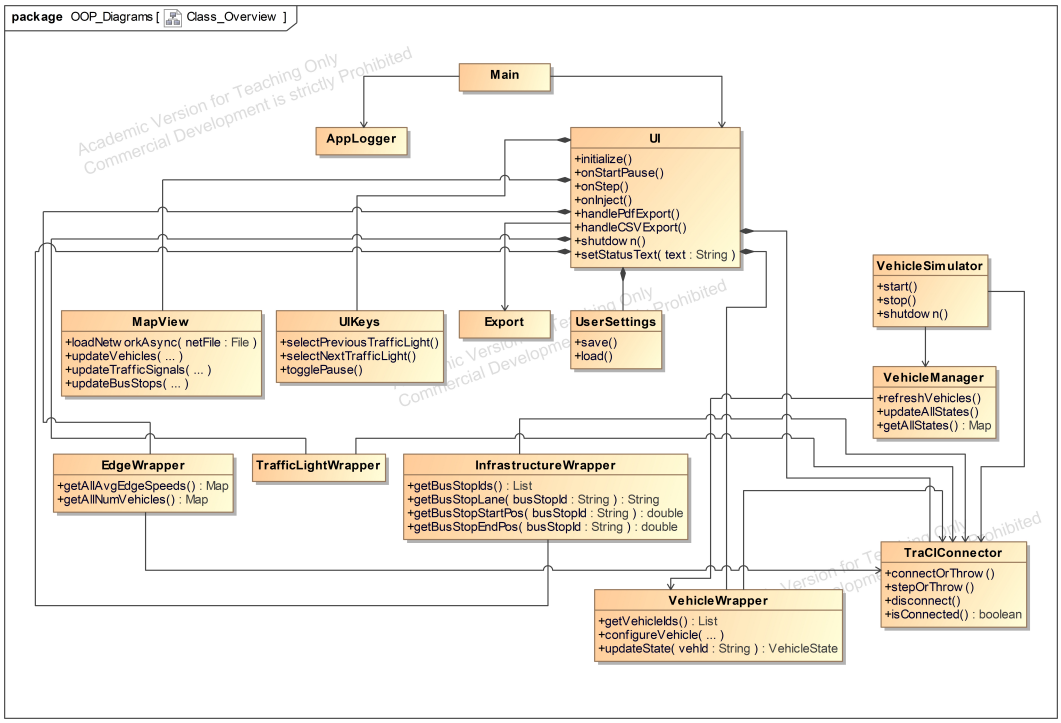


Figure 3.19: Class Diagram Overview

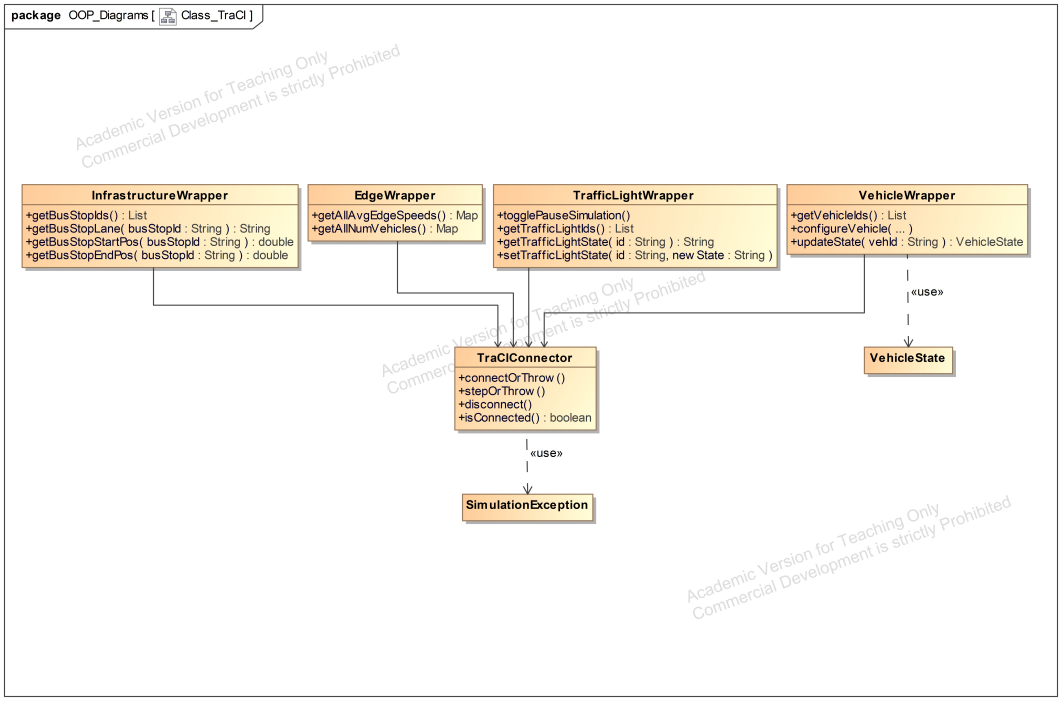


Figure 3.20: TraCI Class Diagram

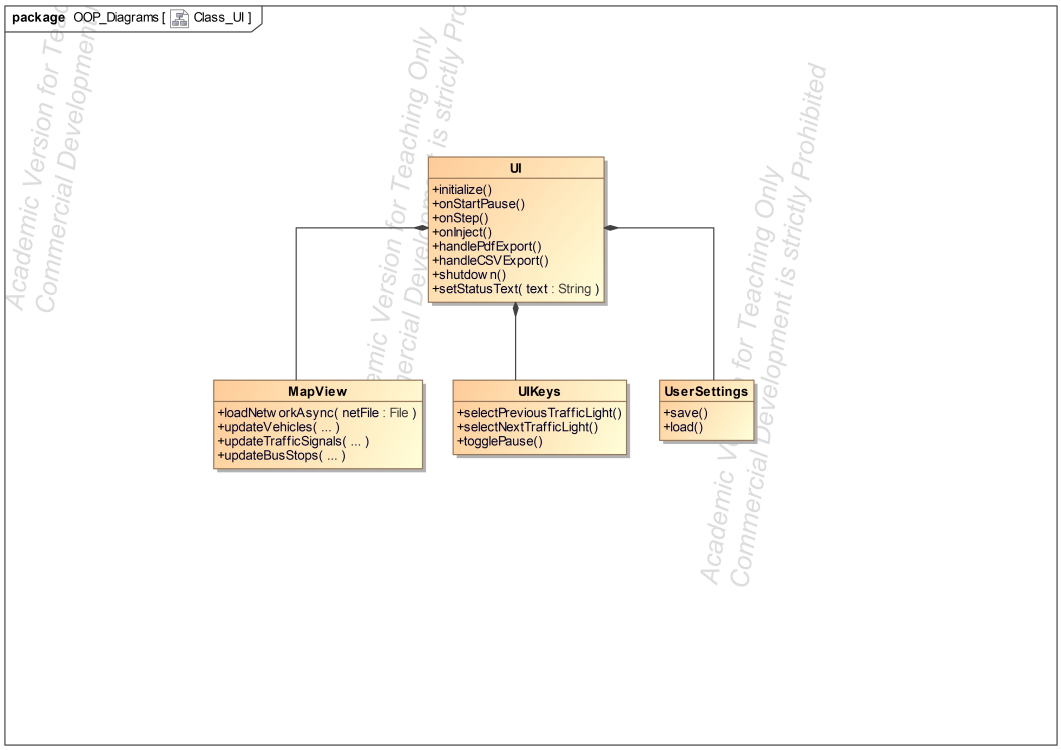


Figure 3.21: UI Class Diagram

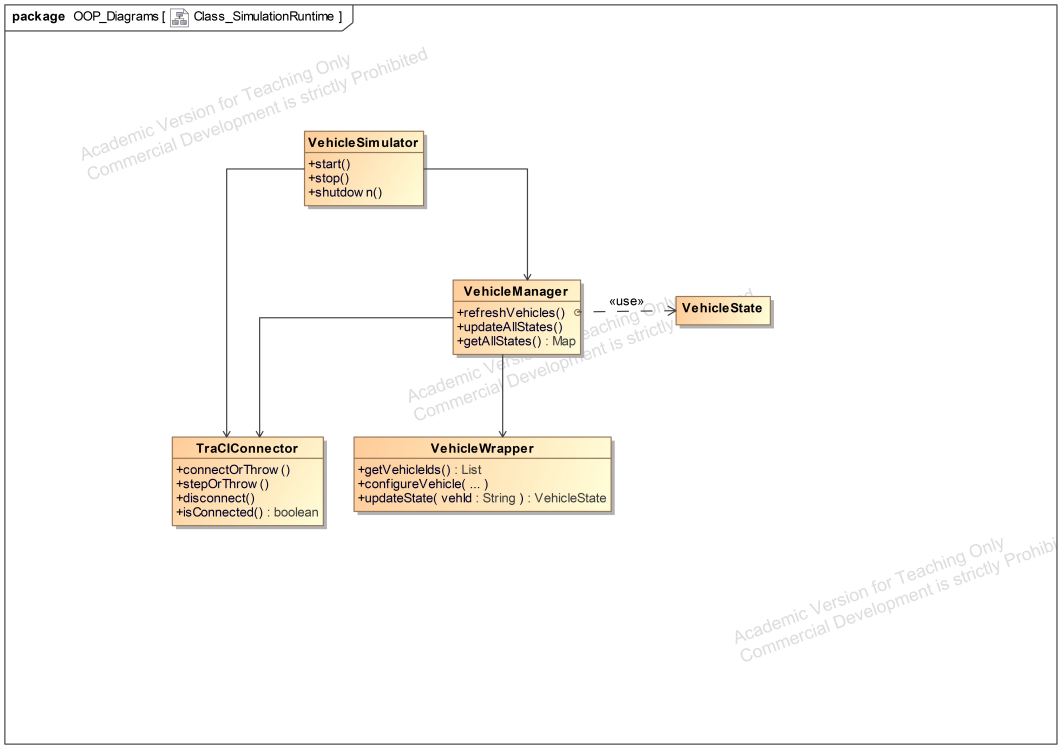


Figure 3.22: Simulator Class Diagram

The `TraCIConnector` class forms the core of the system, responsible for establishing a connection to SUMO, tracking the simulation step, and providing basic operations such as `connect()`, `step()`, and obtaining simple vehicle counts. The `VehicleWrapper` and `TrafficLightWrapper` classes each hold a reference to this connector and serve as early abstractions for accessing SUMO vehicle data and traffic light information. The `Main` class simply launches the application. This structure creates a clean separation of concerns and prepares the system for more advanced TraCI functions.

3.5 Use Case Diagram

Figure 3.23 presents the simplified use case diagram.

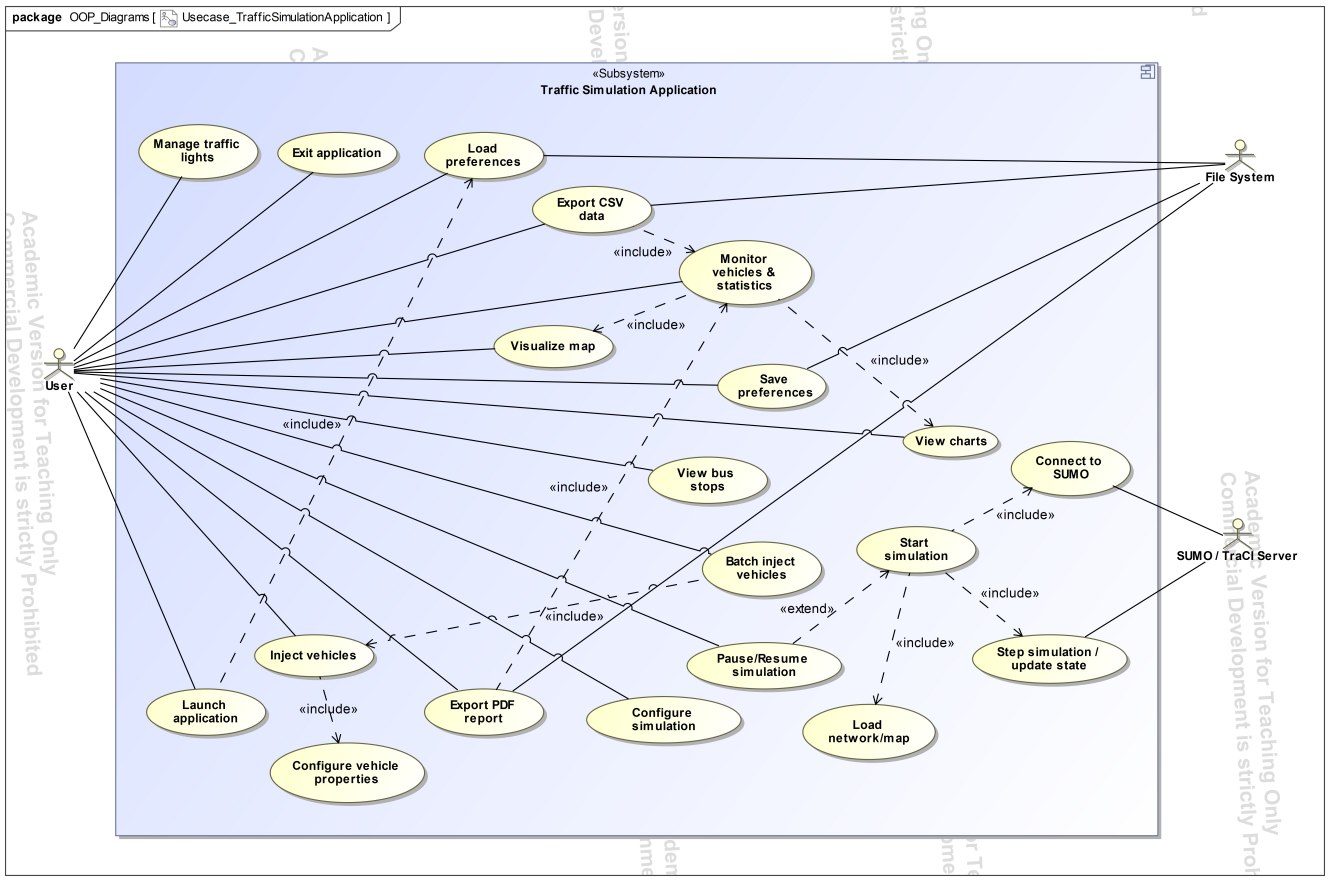


Figure 3.23: Use case diagram for the traffic simulation application

Most of the functionality here is outlined in the Core Features section. Please refer to it for more information.

3.6 Code Documentation and User Guide

Core components are documented using Javadoc and targeted inline comments, focusing on public APIs and non-trivial logic (e.g., TraCI connection lifecycle, map rendering, and vehicle state caching). A draft user guide (`userguide.md`) describes environment setup, launching the application, and the main UI workflows.

General documentation of the project can be found at:
`https://github.com/hunggiadao/realtime-traffic-simulation-java-oop/blob/main/README.md`

The user guide can be found at:
`https://github.com/hunggiadao/realtime-traffic-simulation-java-oop/blob/main/userguide.md`

3.7 Stress Test Scenario and Results

To validate the prototype under load, we prepared a stress-test scenario (`Stress.sumocfg`) that generates a high number of vehicles to stress vehicle injection, filtering, and rendering performance. This scenario was used to identify CPU-intensive parts of the update loop (map redraw and table refresh) and to motivate throttling strategies (e.g., updating the table less frequently or increasing the simulation `step-length`).

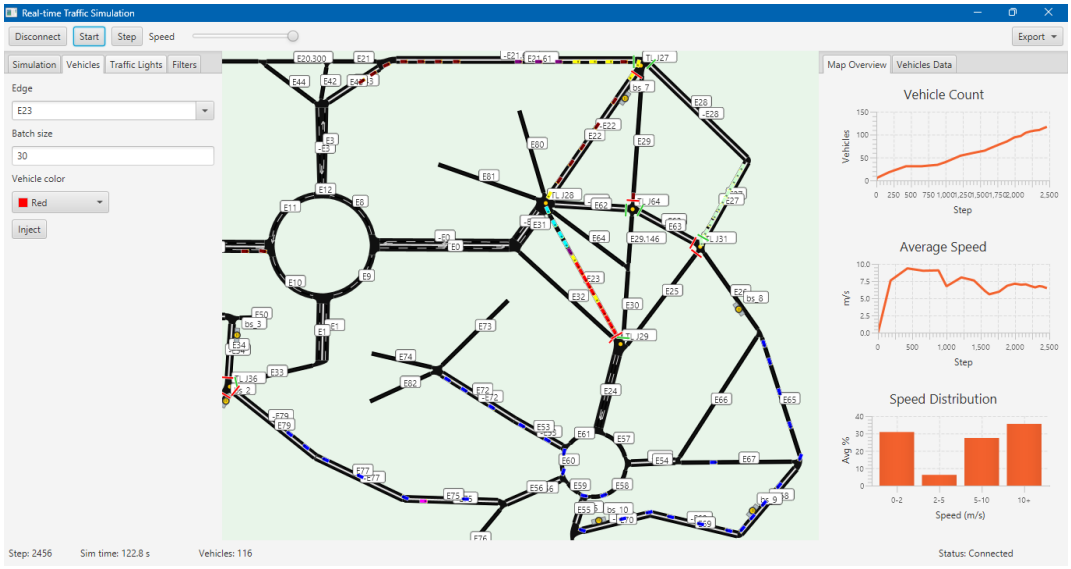


Figure 3.24: Stress test run of the application with a high vehicle count.

3.8 Traffic Network Preparation

To test the SUMO setup during Milestone 1, we constructed a small traffic network based on a real street layout in Frankfurt. The area was selected for its moderate size and clear road geometry, making it ideal for early simulation testing. Figure 3.25 shows the reference map used during the process.

Figure 3.25: Reference street layout used for building the SUMO network.

Using SUMO's NETEDIT tool, the roads, intersections, and lane connections were recreated manually. This allowed the team to produce a valid `.net.xml` file that could be loaded into SUMO for simulation.

3.9 SUMO Network and Initial Simulation Test

Once the road geometry was created in NETEDIT, the network was paired with a basic route file and loaded into SUMO for testing. Figure 3.26 shows the SUMO GUI rendering of the test network.

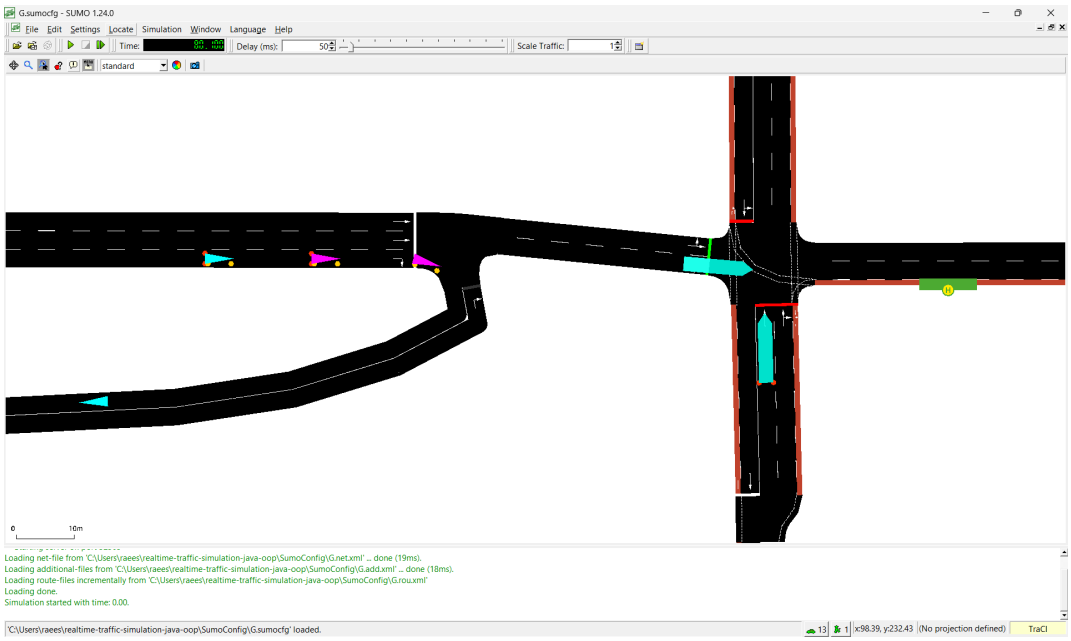


Figure 3.26: SUMO network used for the initial simulation test

This simulation confirmed that lane connections, turning paths, and traffic flow behaved as expected. The successful run ensured that the SUMO environment, map setup, and route files were correctly prepared for Java integration in later phases.

3.10 TraCI Connection Test

During Milestone 1, we performed a small integration test to verify that a Java application could successfully communicate with SUMO via TraCI.

The connector established a SUMO process, stepped the simulation once, and retrieved a simple vehicle count, confirming a working communication pipeline. Figure 3.27 illustrates the demo used during testing.

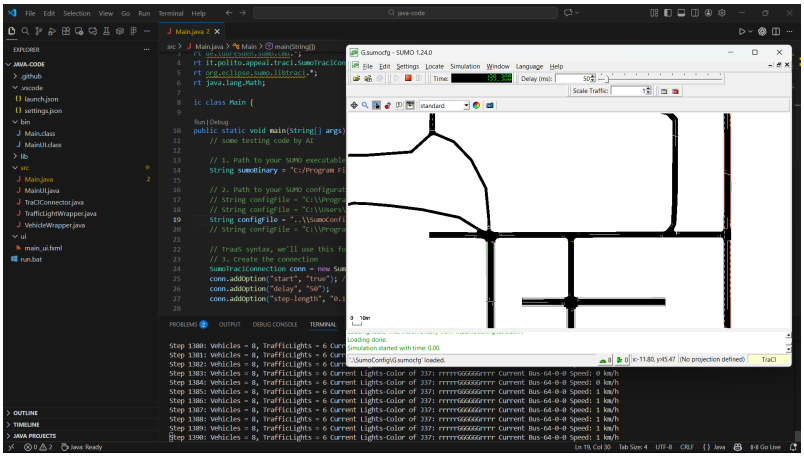


Figure 3.27: Demo of the Java–SUMO TraCI connection during Milestone 1.

This confirms that the communication layer works and can be expanded with more detailed functions in the next milestones.

Conclusion

3.11 Conclusion

By the final milestone, we have delivered a fully functional realtime traffic simulation application. The system can connect to a live SUMO simulation via TraCI, run or step the simulation, visualize the network and moving vehicles directly in the JavaFX GUI, and export simulation data for further inspection and fine tuning. Core interactive features are implemented, enabling users to actively influence traffic flow during runtime.

We have also addressed performance challenges presented during Milestone 2. The main challenges identified are performance and update-frequency trade-offs when handling large vehicle counts, particularly during stress testing. These findings guide the next development steps: improving rendering efficiency, reducing unnecessary UI refresh work, and extending the control logic. By employing lazy updates and utilizing as many dedicated SUMO function calls as possible to reduce computing overhead, we achieved better running performance and less lag time than we had had in Milestone 2.

For a performant application, we relied primarily on the built-in SUMO libraries, which provide efficient and easy to use value retrievals and setting methods. We also eliminated most performance issues encountered in Milestone 2 by employing lazy updates or caching of commonly used values. Overall, our application is significantly more responsive than before. As a result, this allowed for more fine-tuned rendering and a smaller `step-length` for smooth animation.

3.11.1 Performance Evaluation

To address the bottlenecks identified during stress testing, we plan to:

- **Optimized map rendering**: Implement spatial partitioning (e.g., QuadTree) to only render vehicles and map elements currently within the viewport.
- **Separated the threads between map simulation and UI rendering**: to maintain simulation as fast as possible, and only update the UI when needed.
- **Throttled UI updates**: Decouple the data fetch rate from the render rate to ensure the UI remains responsive even when the simulation is running at high speeds.

3.11.2 Advanced Traffic Control

We have extended the current manual control capabilities with automated logic:

- **Automated Traffic Light Management**: Implement algorithms that dynamically adjust signal phases based on real-time queue lengths detected by the simulation.
- **Vehicle Grouping**: Allow users to define groups of vehicles and apply batch commands (e.g., "reroute all buses").

3.11.3 Team Reflection

In retrospective, there were clear challenges for us during the duration of this project. Over time, we had worked continuously to improve our work and productivity, though some difficulties were solved and others were not.

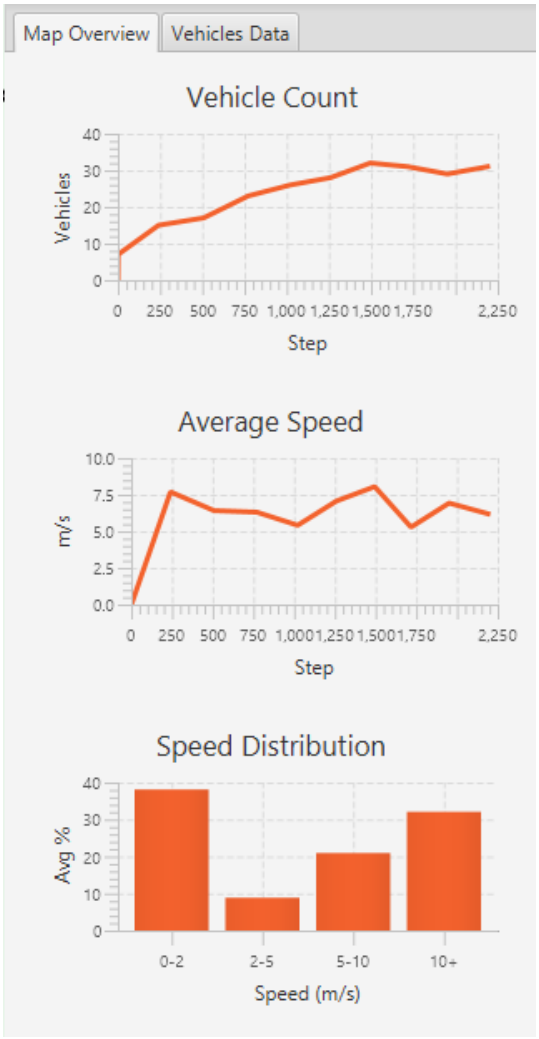
Our group formed later than other groups in the project, so we were initially behind in terms of progress, but still managed to achieve all requirements for each milestone. We also had a good starting structure, by organizing our Github repository to be robust and modular. In this manner, we could maximize the amount of work being done simultaneously by assigning each member a separate non-overlapping part of the project. We maintained a commit history in detail and employed good coding practice (branching from main, pulling and pushing) so that we could minimize time spent to debug, fix errors, and resolve Git merge conflicts.

During the later parts of our project, there was a shift in participation among some team members. Some members who had been active in the beginning did not participate in the final stage, and some contributed significantly more than they had done previously. This did not drastically affect our code production or productivity, but it shifted our communication to other team members. We have tried to contact the non-participating members via group messaging and direct messaging, but they did not respond timely or help in committing new code to the repository. Despite this difficult, we managed to complete the application with a sophisticated set of features and absence of most bugs.

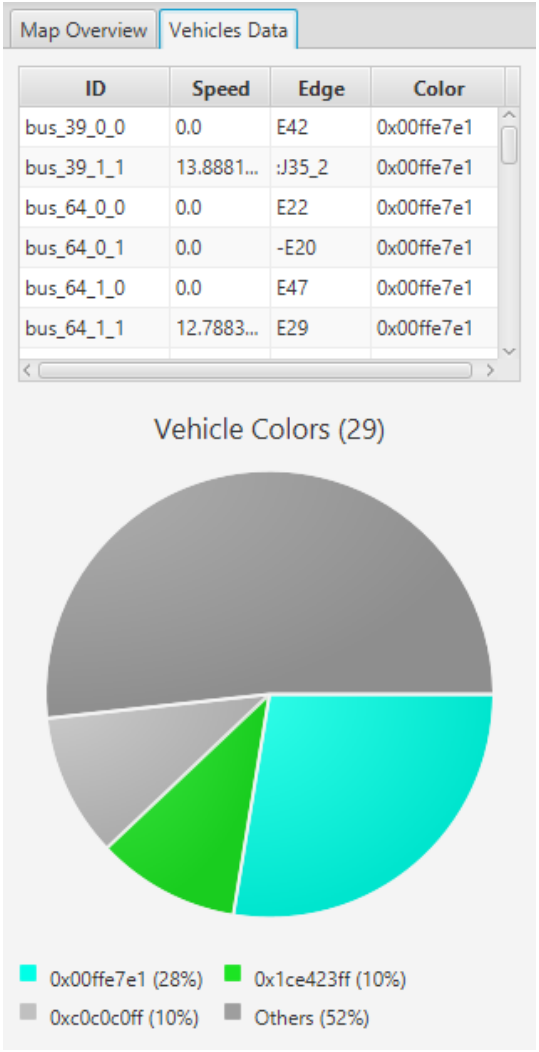
3.11.4 Reporting and Analytics

The final application will include comprehensive reporting tools:

- **Data Export**: Functionality to export simulation statistics (travel times, congestion levels) to CSV or PDF formats.
- **Enhanced Dashboard**: A more detailed statistics view including average waiting times and other estimates.



(a) Map Overview metrics panel



(b) Vehicles Data metrics panel

Figure 3.28: Layout for the advanced analytics dashboard