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

1651339 Dang, Huu Trung Son

1651313 Pham, Khac Uy

Contents

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 Milestone 2, the project has reached a functional prototype stage. The application can establish a live SUMO connection, advance the simulation in real time (or step-by-step), and render the SUMO network directly inside the JavaFX GUI with moving vehicles. 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). 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 Milestone 2 deliverables include inline code documentation (Javadoc), a draft 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.

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.

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 Milestone 2 Status

Milestone 2 focused on delivering a functional prototype with core features implemented. The current status is summarized below.

- **Working Application:** Live SUMO connection, map visualization inside the GUI, vehicle injection, and traffic light control are implemented.
- **Code Documentation:** Core classes contain Javadoc and inline comments (e.g., connection, vehicle, and map rendering components).
- **User Guide Draft:** A draft user guide is provided (`userguide.md`) describing setup and basic usage.
- **Test Scenario:** A stress scenario configuration (`Stress.sumocfg`) is provided to test high vehicle counts.
- **Continuous Integration:** A CI pipeline was established to automate build and test processes, ensuring code stability.
- **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 31st December
- **Traffic Light Adaptation:** Manual and rule-based control system by 31st December
- **Exportable Reports:** CSV and PDF export functionality by 31st December
- **Final Documentation:** Updated user guide and comprehensive code documentation by 10th January
- **Project Summary:** Enhancements overview and design decisions report by 10th January

2.2 Task Distribution

Team roles were revisited and refined during Milestone 2 to match the implemented components.

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

2.3 Challenges and Risks

- **Performance under load:** With many vehicles, frequent UI refreshes (map + table) can become CPU-intensive.
- **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.

2.4 Milestone 2 Progress Summary

Milestone 2 focused on converting the Milestone 1 UI into a functional prototype with end-to-end SUMO integration. The following summary highlights the status and main outcomes.

- **Status:** Core prototype features completed (live SUMO connection, in-app map rendering with moving vehicles, vehicle injection, and traffic light control).
- **Key changes:** Added a real-time stepping loop, integrated vehicle state retrieval for the table and map, added injection controls (count/speed/color), and implemented traffic light phase switching and duration adjustments.
- **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.

2.4.1 Software Engineering Practices (Git)

- **Branching + PRs:** Development was performed on feature branches and merged 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*) and describe the affected subsystem (UI, simulation, wrappers, build).
- **Traceability:** Milestone 2 work is visible through incremental commits that introduce the simulator loop and vehicle-management abstractions, followed by bug fixes and UI integration improvements.

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 Scene Builder. 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.

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.

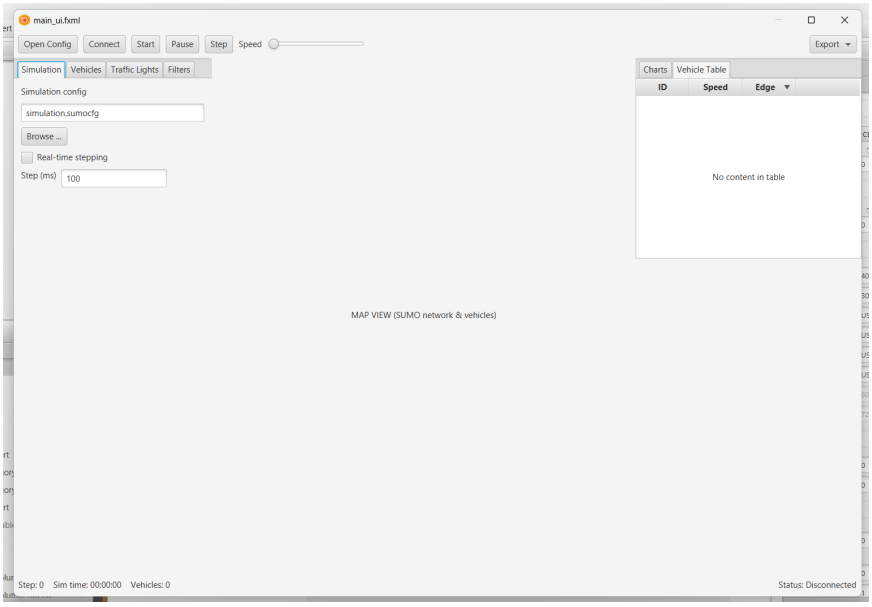

Figure 3.1: Graphical User Interface designed for Milestone 1

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.

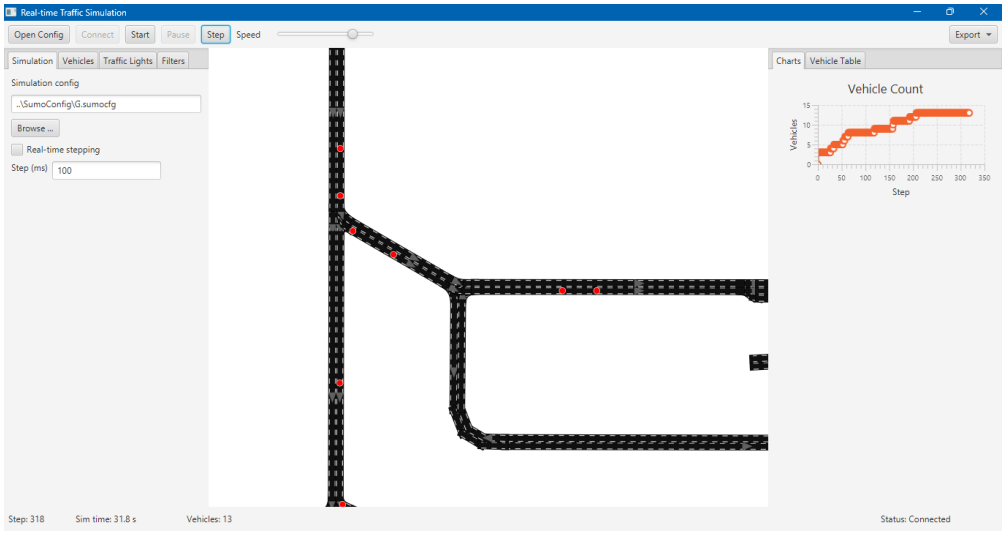

Figure 3.2: Milestone 2 functional prototype: live map rendering and simulation controls in the JavaFX GUI.

The central region contains the map view. In Milestone 2, this region renders the SUMO network (parsed from the network file referenced by the loaded `.sumocfg`) and draws moving vehicles based on TraCI position updates. The map supports zooming and panning to inspect local behavior around intersections.

On the right side, a second `TabPane` provides simulation metrics. The *Vehicle Table* tab contains a JavaFX `TableView` that is populated with live data (vehicle ID, speed, and current edge). A simple line chart tracks vehicle count over time.

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

Overall, the UI now provides a complete workflow for Milestone 2: load a scenario, connect to SUMO, run or step the simulation, visualize the network with moving vehicles, inject vehicles interactively, and manipulate traffic light phases.

3.2 Milestone 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, and the GUI reflects the current state through a status bar (step number, simulation time, and vehicle count).

3.2.2 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.

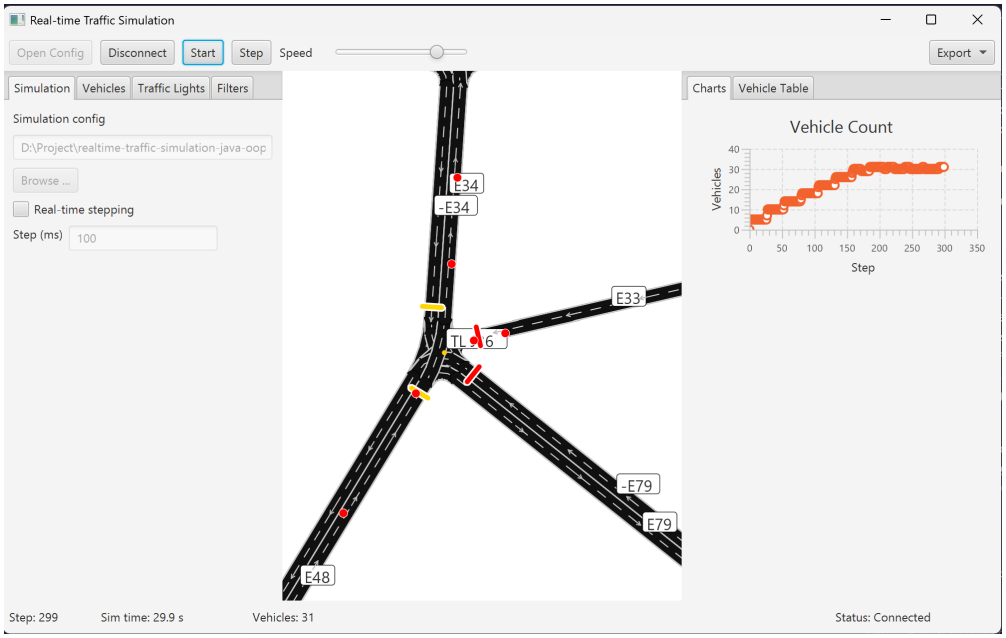

Figure 3.3: Zoomed-in map view with moving vehicles and traffic-light stop-line overlays (red/yellow/green).

3.2.3 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.

3.2.4 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.

3.2.5 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.

3.3 Technical Implementation

To support the functional requirements of Milestone 2, 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.

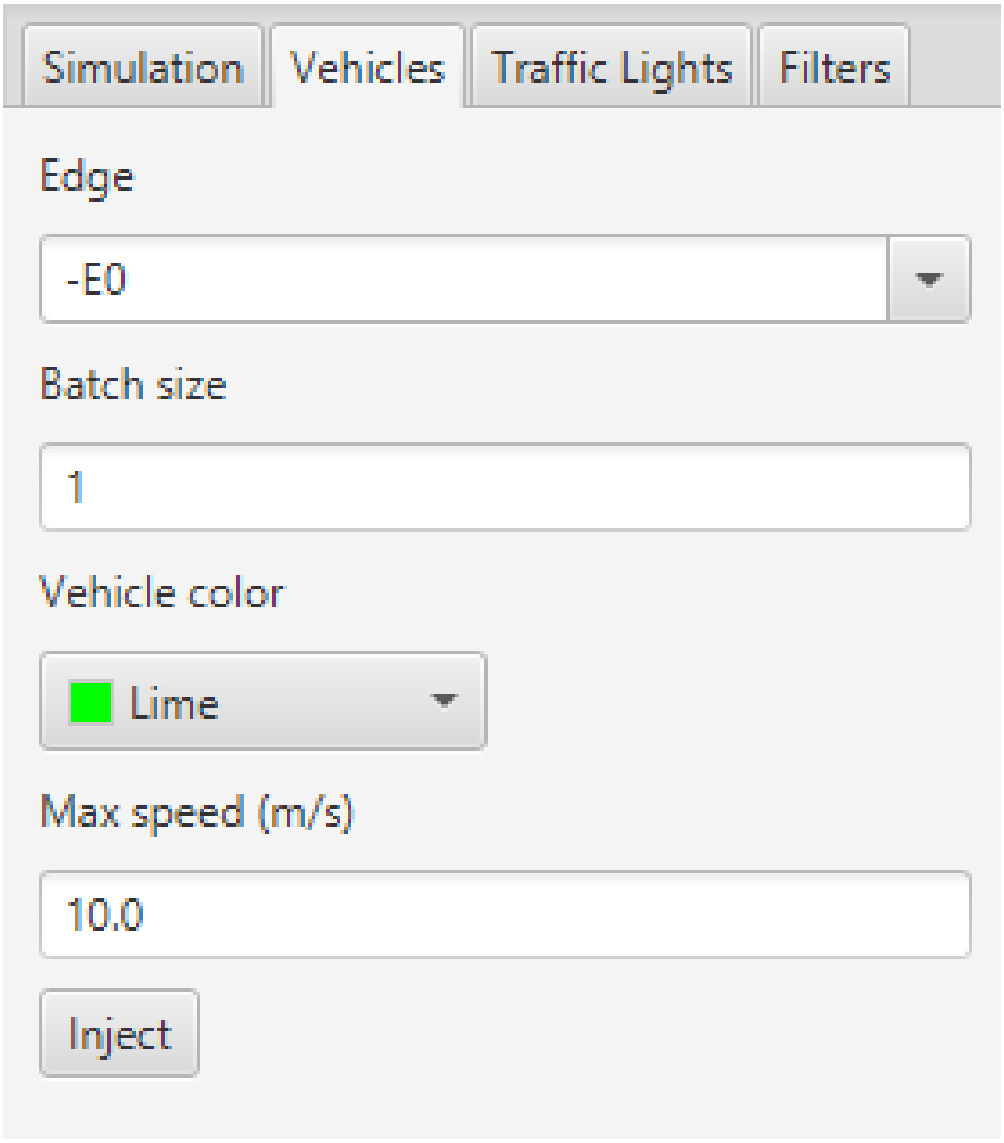

Figure 3.4: Vehicle injection controls used during stress testing, including color selection.

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

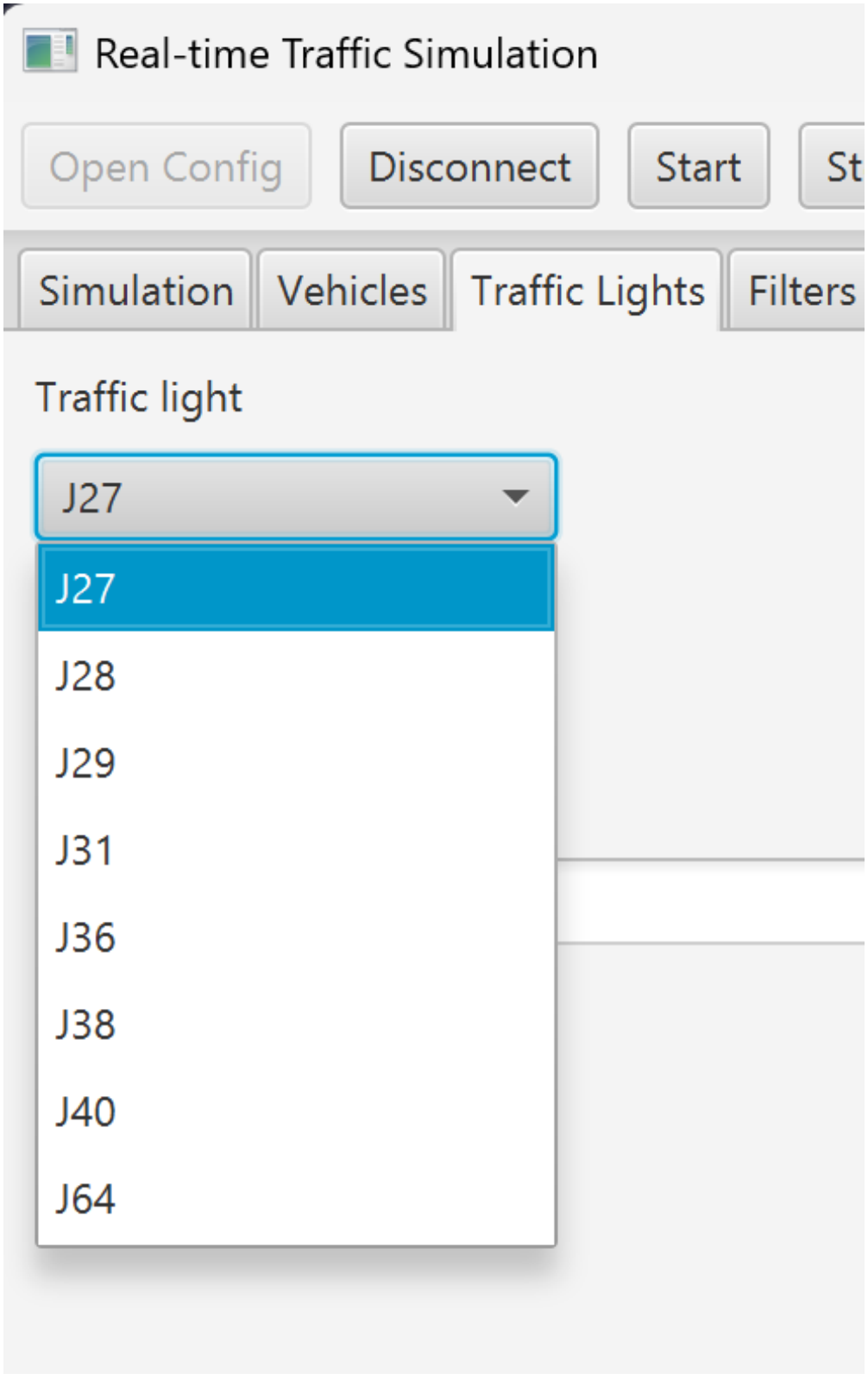

Figure 3.5: Traffic light selection and manual control: phase/state display and phase duration adjustment.

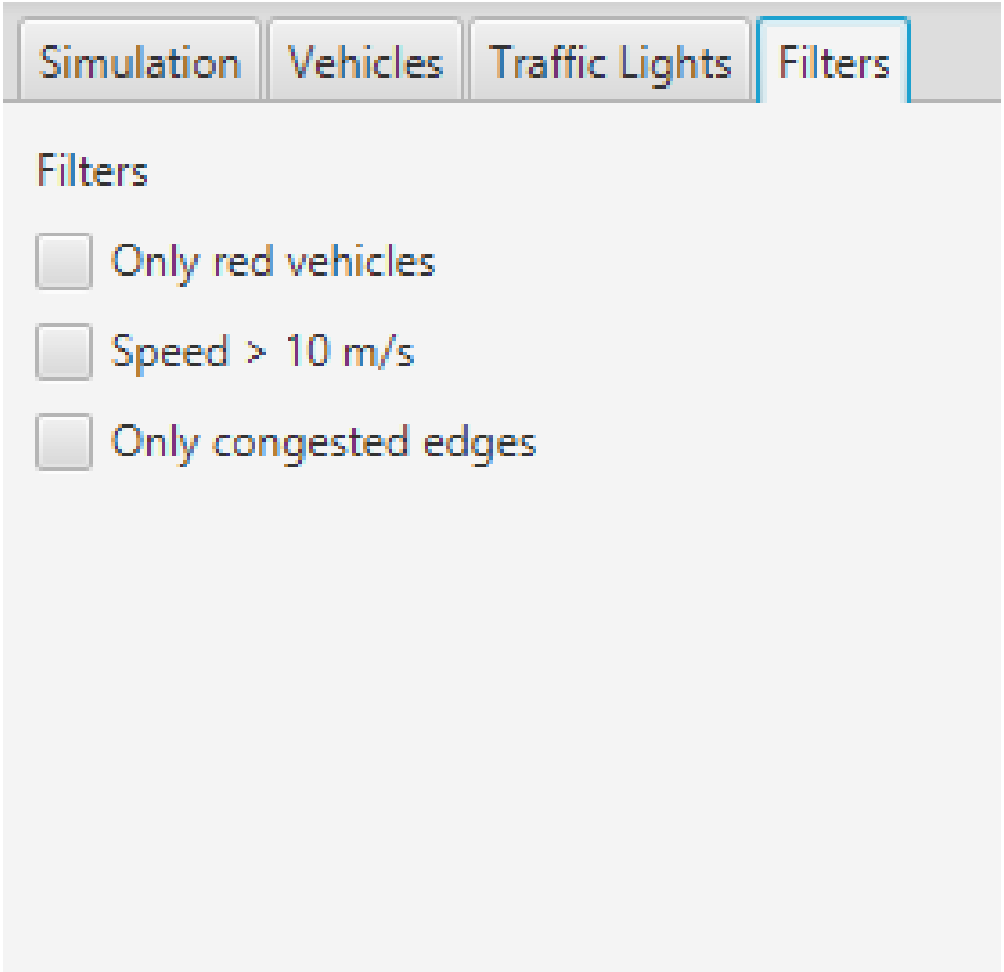

Figure 3.6: Filter controls used to isolate vehicles by color, speed, and congestion heuristics.

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.

3.3.3 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 critical information is always visible on top.

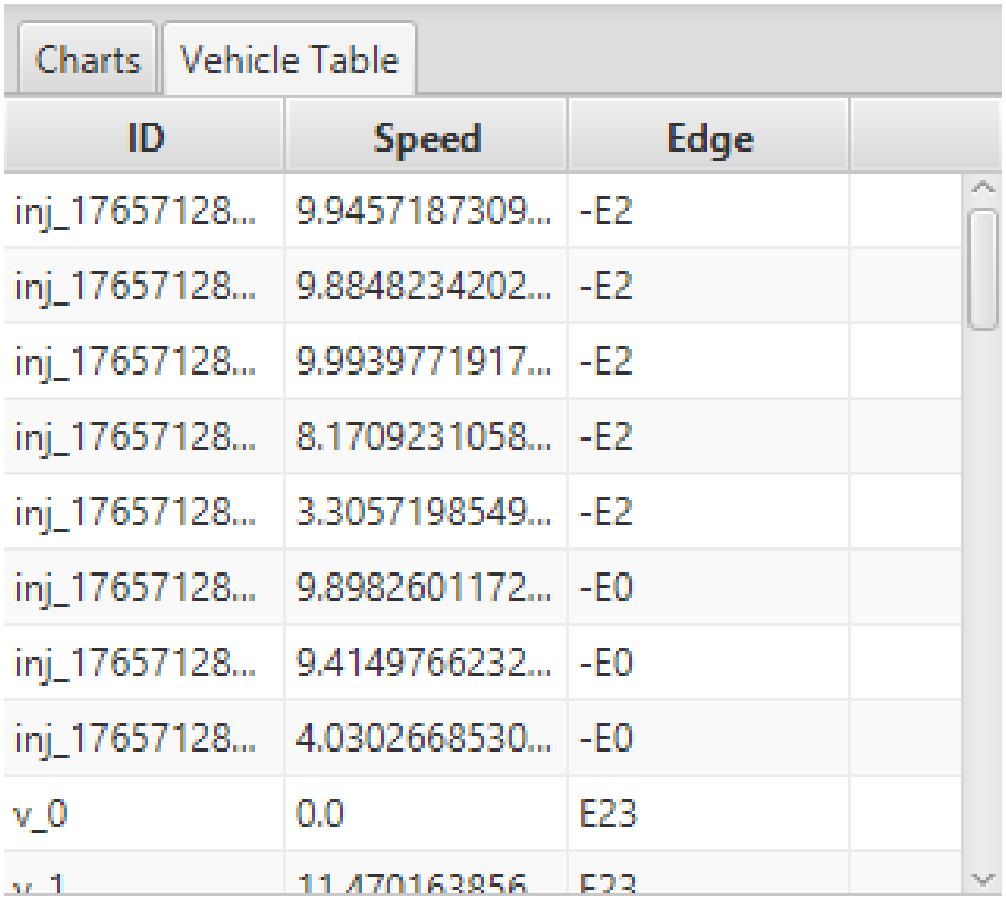

Figure 3.7: Vehicle table populated with live simulation data (filtered view).

3.3.4 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.

3.3.5 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`.
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.

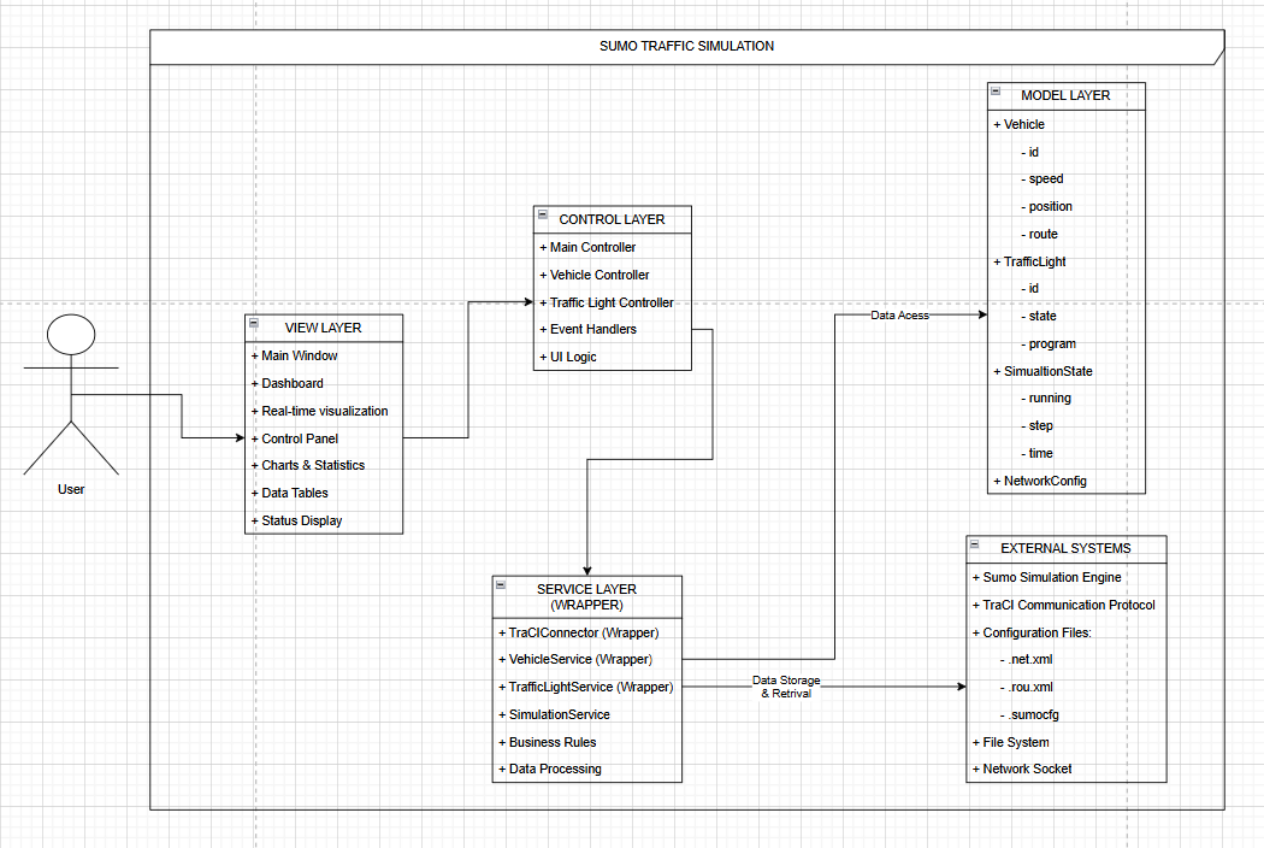

Figure 3.8: System Architecture Diagram illustrating the layered design and data flow between the JavaFX GUI, Controller, Wrapper, and SUMO.

3.4 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).

3.5 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.

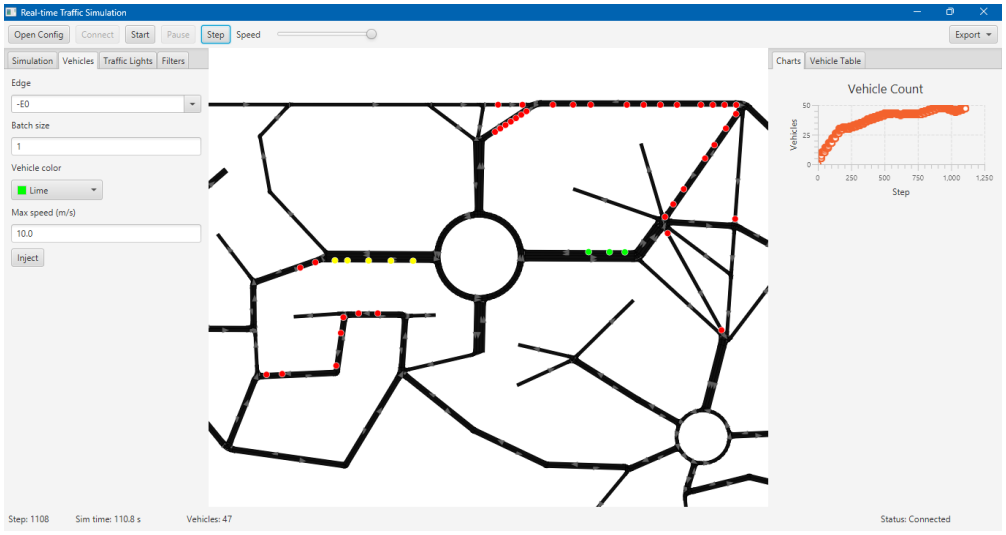

Figure 3.9: Stress test run of the application with a high vehicle count.

3.6 Class Diagram Overview

Figure 3.10 shows the simplified class structure implemented for Milestone 1. The architecture is intentionally minimal, as the purpose of this stage is to establish clear responsibilities before full SUMO integration is added.

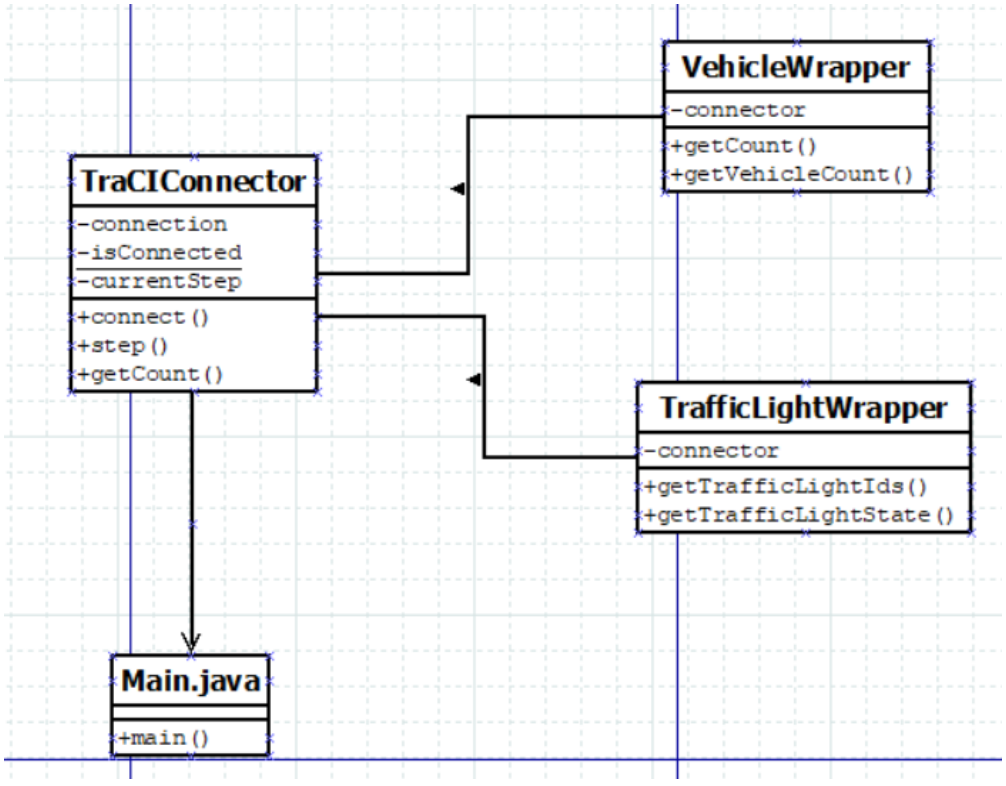

Figure 3.10: Class diagram used in Milestone 1.

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 in future milestones.

3.7 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 3.11 shows the reference map used during the process.

Figure 3.11: 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.

3.8 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 3.12 shows the SUMO GUI rendering of the test network.

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.

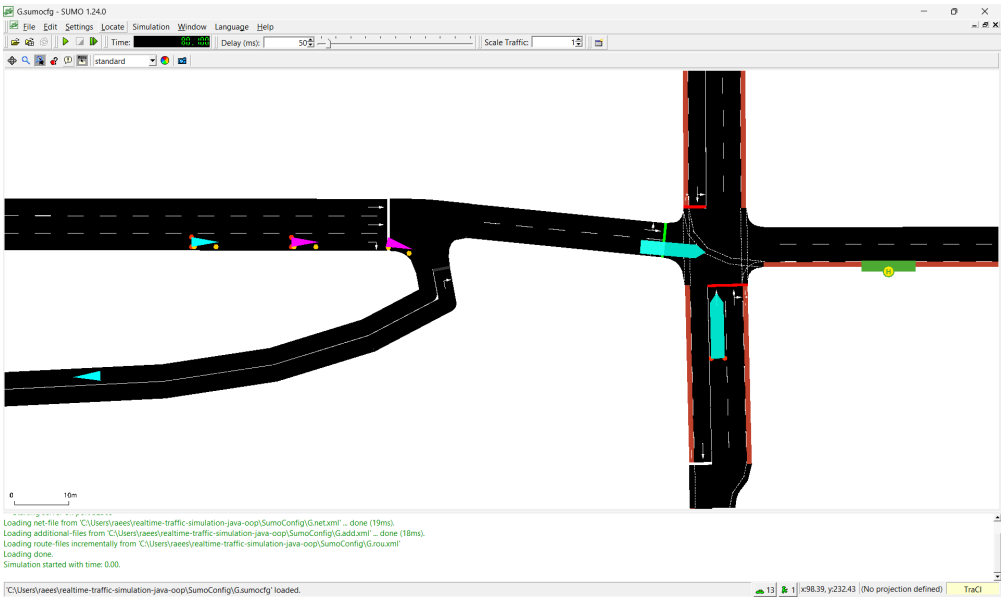

Figure 3.12: SUMO network used for the initial simulation test.

3.9 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 3.13 illustrates the demo used during testing.

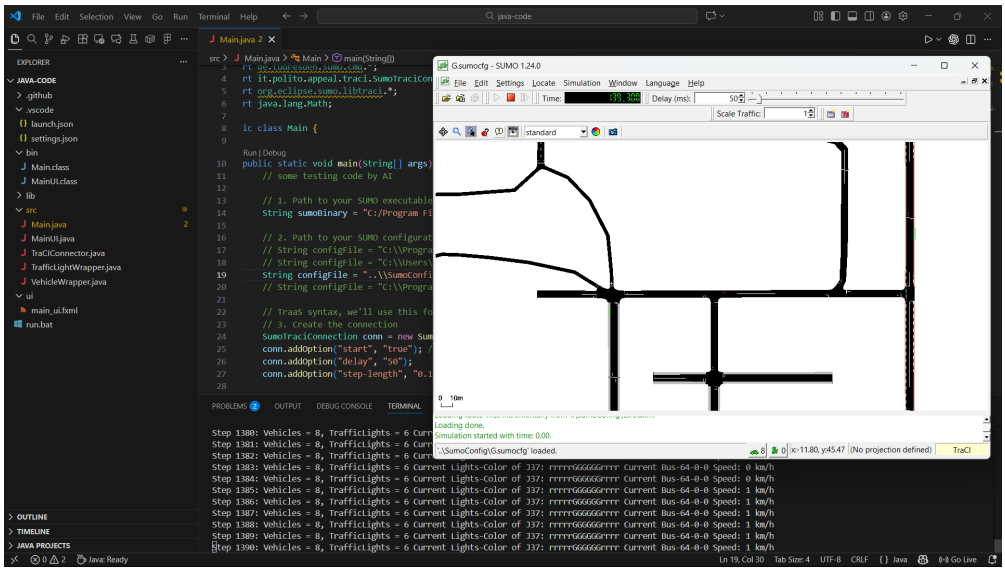

Figure 3.13: Demo of the Java–SUMO TraCI connection during Milestone 1.

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

3.10 Use Case Diagram

Figure 3.14 presents the simplified use case diagram created for Milestone 1.

Since the full system logic is not yet implemented, the current use cases focus on core interactions: running the SUMO simulation, stepping the simulation manually, obtaining simple vehicle counts, and displaying results either in the GUI or console.

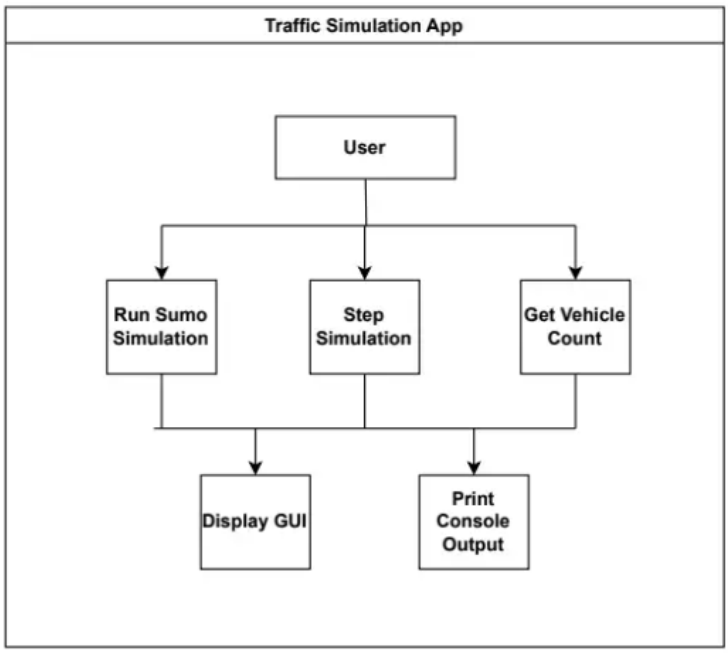

Figure 3.14: Use case diagram for the early-stage traffic simulation application.

These use cases provide a clear overview of what functionality is available at this stage and what interactions will be expanded in Milestone 2.

Conclusion and Future Work

3.11 Conclusion

By Milestone 2, we delivered a functional prototype of the real-time traffic simulation application. The system can connect to a live SUMO simulation via TraCI, run or step the simulation, and visualize the network and moving vehicles directly in the JavaFX GUI. Core interactive features—vehicle injection and manual traffic light control—are implemented, enabling users to actively influence traffic flow during runtime.

The main challenges identified at this stage 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. Overall, Milestone 2 establishes a solid, end-to-end pipeline from SUMO to a responsive GUI with tangible control capabilities.

3.12 Future Work

Building on the functional prototype, the final phase of development will focus on optimization, advanced features, and user experience refinements.

3.12.1 Performance Optimization

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

- **Optimize Map Rendering:** Implement spatial partitioning (e.g., QuadTree) to only render vehicles and map elements currently within the viewport.
- **Throttle 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.

3.12.2 Advanced Traffic Control

We will extend 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").

3.12.3 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 CO2 emission estimates.

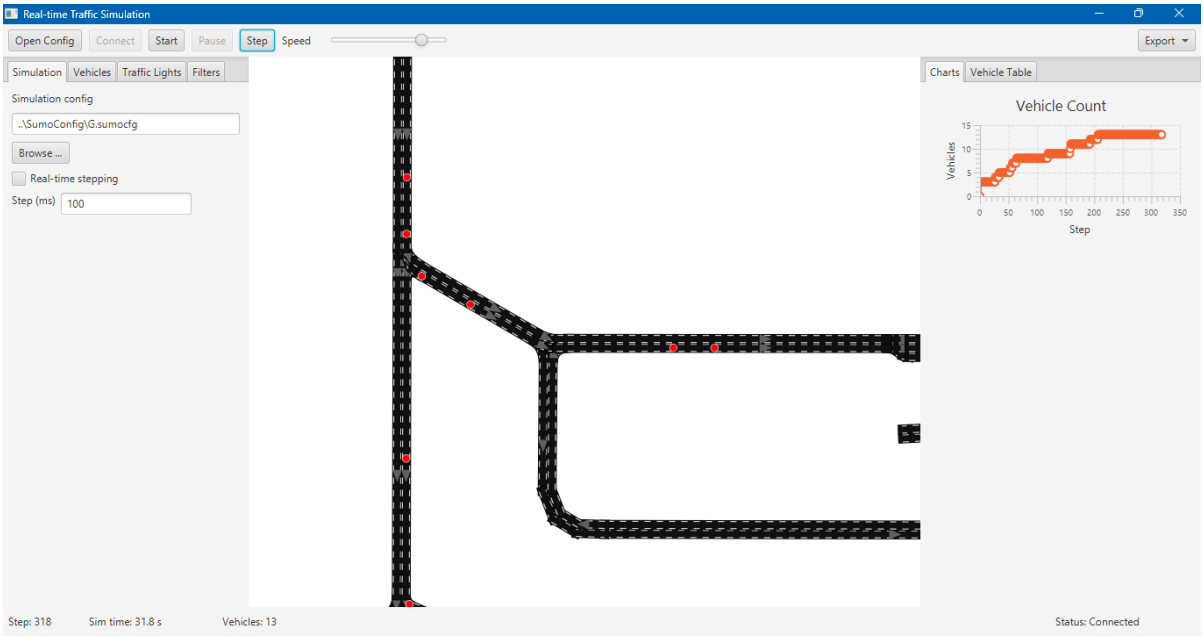

Figure 3.15: Proposed layout for the advanced analytics dashboard. Note the 'Export' button (top right) and the expandable 'Charts' pane, which will be fully implemented in Milestone 3.