Frankfurt University of Applied Sciences

– Faculty of Computer Science and Engineering –

Real-Time Traffic Simulation with Java

Final Project Documentation

Module: Object-Oriented Programming in Java
Winter Semester 2025/2026

Submitted on 18.01.2026 by

Mauricio de Souza Hilpert,

Elias Heß,

Yiyuan Ma,
Salaheddine Mabchour,

Laura Gradwohl

Source Code Repository:
https://github.com/33meow/RealTimeTrafficSimulation/tree/main

Lecturer: Prof. Dr.-Eng. Ghadi Mahmoudi

1. Introduction
1.1. Project Overview and Objectives
The objective of this project is to develop a Real-Time Traffic Simulation application using Java and the SUMO (Simulation of Urban MObility) engine. The system is designed to provide a graphical interface for monitoring traffic flow, controlling traffic lights, and analyzing environmental data (CO2 emissions) in real-time.
The application serves as a bridge between the raw simulation data provided by the TraaS API and the user, offering features such as:

- Interactive Visualization: A custom-rendered map displaying roads, vehicles, and traffic light phases.
- Dynamic filtering: Allowing users to isolate specific vehicles based on criteria (e.g., street or type).
- Stress Testing: A dedicated multi-threaded module to simulate high-load traffic scenarios.
- Data Export: Generating CSV reports for post-simulation analysis.

1.2. Summary of enhancements and design decisions

The goal for the last milestone was a fully working application with all the required features that is easy to use and to have a user guide that makes it easy to understand what our application can do.
For our final milestone we added filters to make it possible to group our vehicles and decide which vehicles should be visible and we also added filters to select street edges.
Additionally it is now possible to download a CSV file with the log entries for the runtime of the application to ensure the automatic generation of output files in a consistent structure.
Each new feature is accessible for the end user via our GUI. To keep the project manageable, we decided to use a minimal and modular architecture. The code is separated into clear components. This structure supports readability and makes future extensions easier to implement. We intentionally avoided complex frameworks or heavy external dependencies, because the main objective was to maintain a clean and portable solution that can run on different systems without issues.
Unfortunately we did not reach every goal that we had in mind when we started working on this project. There are some improvements and new features that we would have loved to bring to life but could not during the given time.
For a more detailed insight in what we would have wanted so achieved, see the paragraph under **6.5. Future Works.**
Overall, the project achieves its goal of providing a basic but working Sumo app. The design decisions were made to keep the system simple and extendable, and the missing features represent opportunities for future improvements.

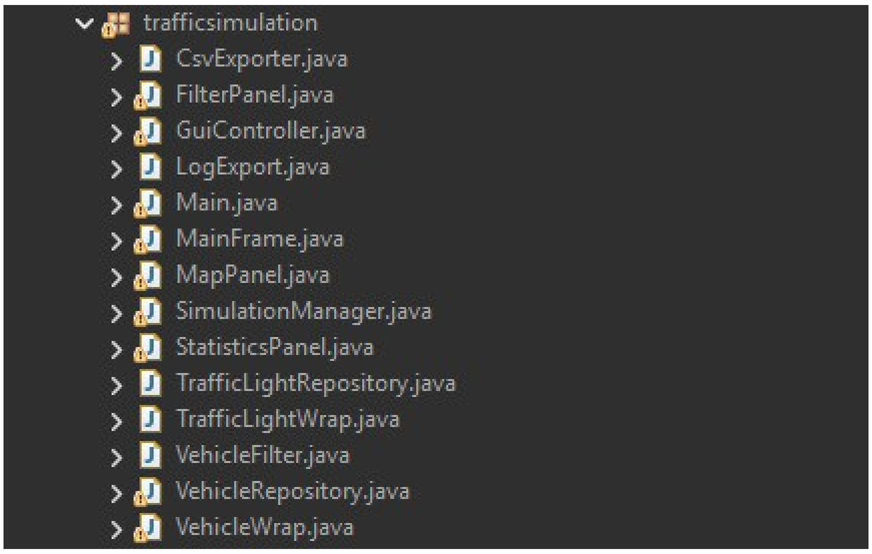
2. System Architecture and Design

This section outlines the architectural decisions and the object-oriented patterns applied to ensure the system is modular, scalable, and thread safe.

2.1. Architectural Pattern (MVC)
The application strictly follows the Model-View-Controller (MVC) pattern to separate the simulation logic from the user interface.

- Model: Represented by SimulationManager, VehicleRepository, and TrafficLightRepository. These classes manage the data state and communicate directly with SUMO.
- View: Represented by MainFrame, MapPanel, and StatisticsPanel. These classes are responsible only for rendering the data to the user.

- Controller: The GuiController class acts as the mediator. It handles user inputs (e.g., "Start", "Stress Test") and updates the Model.



```
trafficsimulation
  CsvExporter.java
  FilterPanel.java
  GuiController.java
  LogExport.java
  Main.java
  MainFrame.java
  MapPanel.java
  SimulationManager.java
  StatisticsPanel.java
  TrafficLightRepository.java
  TrafficLightWrap.java
  VehicleFilter.java
  VehicleRepository.java
  VehicleWrap.java
```

Figure 2.1: Project Package Structure reflecting the MVC components.

2.2. Object-Oriented Design

2.2.1. TraaS API Wrapper Design (Encapsulation)

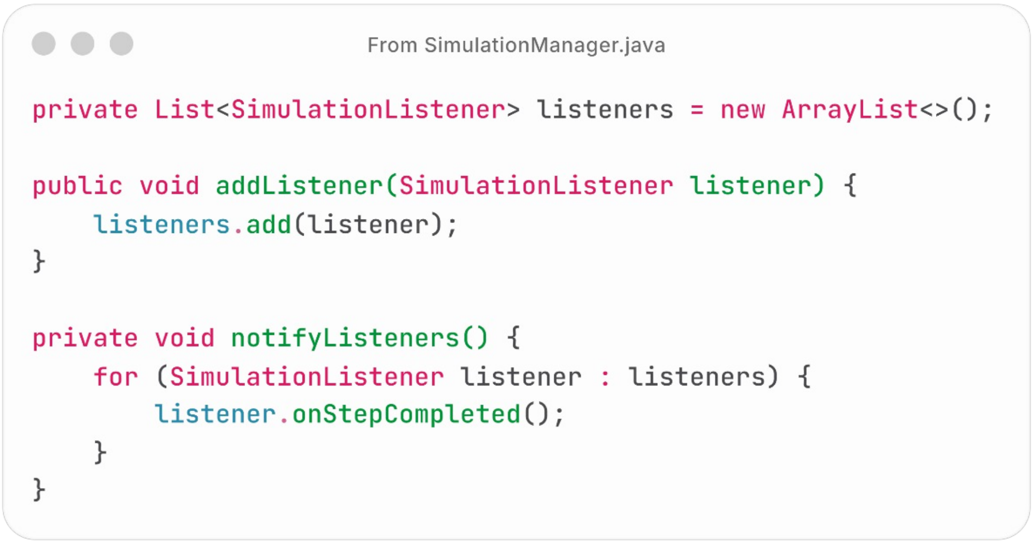
To mitigate the complexity of the procedural TraaS API, we implemented the **Wrapper Pattern**. The classes VehicleWrap and TrafficLightWrap encapsulate the low-level API calls (conn.do_job_get). This ensures that the rest of the application interacts with high-level Java objects rather than raw simulation commands.

From VehicleWrap.java

```
public void updateVehicle() {
    try {
        // Encapsulating raw API calls
        this.speed = (double) conn.do_job_get(Vehicle.getSpeed(id));
        SumoPosition2D pos2D = (SumoPosition2D)
conn.do_job_get(Vehicle.getPosition(id));
        this.position = new Point2D.Double(pos2D.x, pos2D.y);
        // ...
    } catch (Exception e) { ... }
}
```

2.2.2. Observer Pattern (Listeners)

To ensure the GUI updates synchronously with the simulation steps without tight coupling, we implemented a custom **Observer Interface** (SimulationListener). The SimulationManager notifies registered listeners (like GuiController) whenever a simulation step is completed.



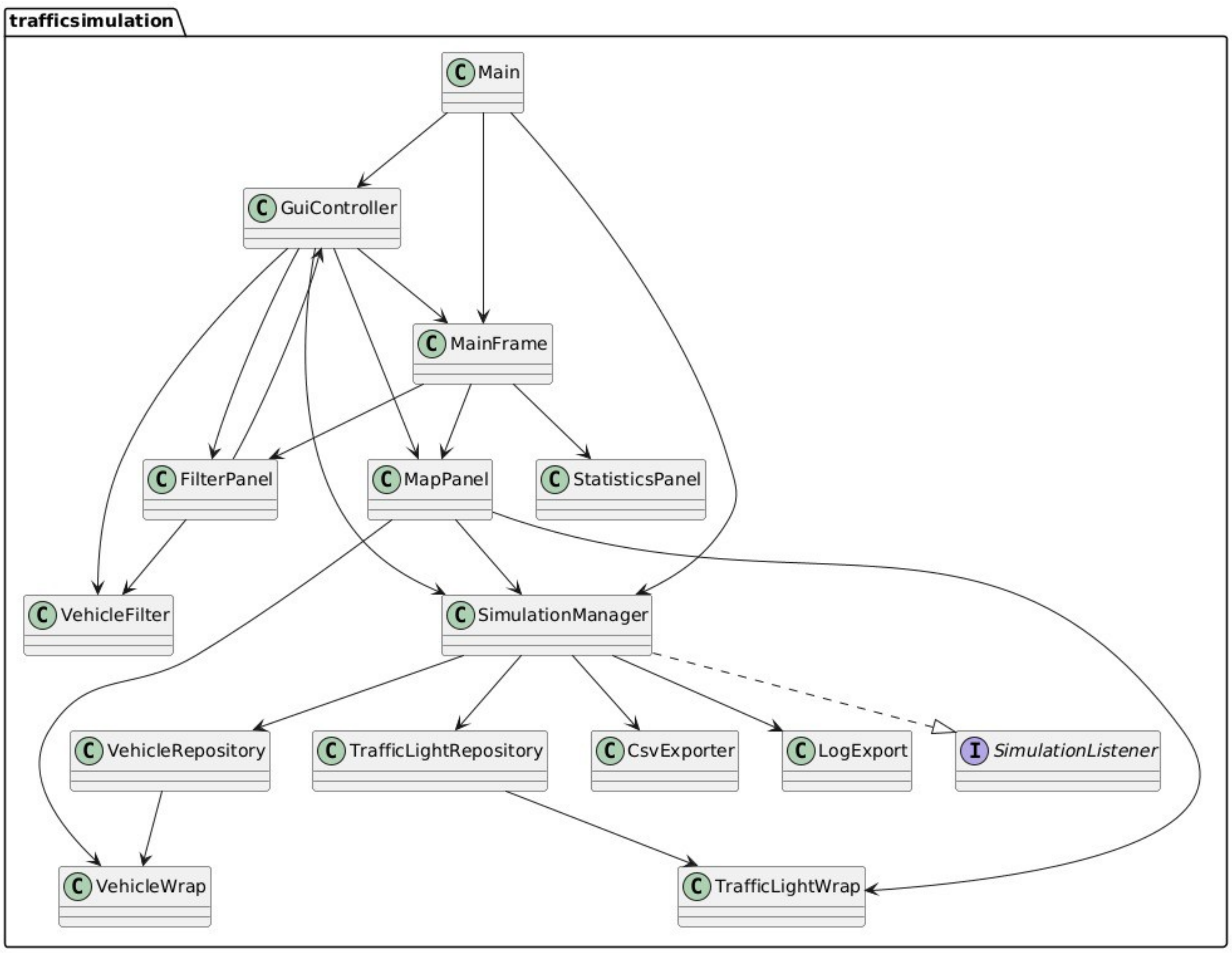
From SimulationManager.java

```java
private List<SimulationListener> listeners = new ArrayList<>();

public void addListener(SimulationListener listener) {
    listeners.add(listener);
}

private void notifyListeners() {
    for (SimulationListener listener : listeners) {
        listener.onStepCompleted();
    }
}
```

2.2.3 Class Diagram (UML)



UML Class Diagram of the Real-Time Traffic Simulation System

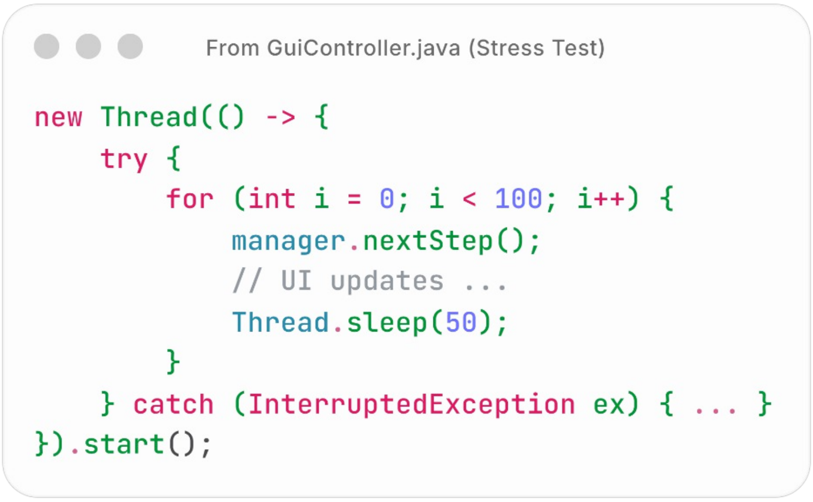
2.3. Data Structures and Collections

- **ArrayList with Synchronization:** In VehicleRepository, we used ArrayList to store active vehicles. To handle concurrency issues during the "Stress Test" (where vehicles are added rapidly), we synchronized the modification methods: public synchronized void addVehicle(...).
- **HashMap for Caching:** In MapPanel, a HashMap<String, Image> is used to cache vehicle textures. This optimization prevents reloading images from disk every frame, significantly improving rendering performance.
- **LinkedList for Statistics:** The StatisticsPanel uses a LinkedList to store the history of CO2 emissions, allowing efficient addition/removal of data points for the live display.

2.4. Design Decisions and Enhancements

Multithreading for Stress Testing

A key design decision was to run the "Stress Test" on a separate thread. Executing heavy simulation loops on the main Swing thread would freeze the UI. By wrapping the logic in a new Thread(), the interface remains responsive while the simulation processes 100+ steps in the background.


From GuiController.java (Stress Test)

```java
new Thread(() -> {
    try {
        for (int i = 0; i < 100; i++) {
            manager.nextStep();
            // UI updates ...
            Thread.sleep(50);
        }
    } catch (InterruptedException ex) { ... }
}).start();
```

Custom Map Rendering

Instead of using a pre-built map component, we extended JPanel in MapPanel and overrode paintComponent(). This allowed us to draw the road network dynamically using Path2D and apply transformations (Zoom/Pan) directly to the graphics context.

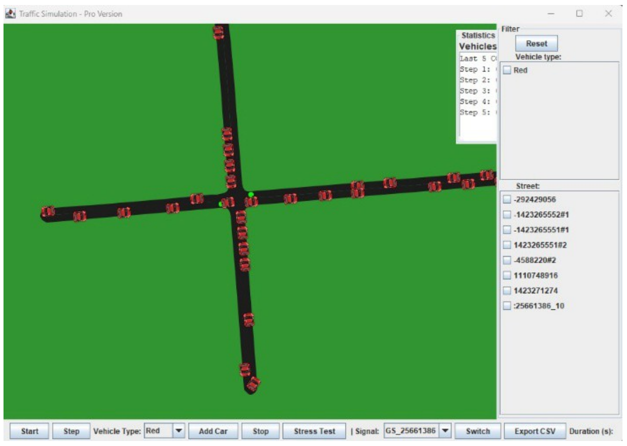


Figure 2.3: Final GUI showing the custom rendered map and real-time statistics.

3. Technical Implementation

This section details the technical realization of the project's requirements, focusing on concurrency, file I/O, error management, and code quality.

3.1. Multithreading and Concurrency

To ensure a responsive user interface (GUI) while performing intensive simulation tasks, the application leverages Java's multithreading capabilities.

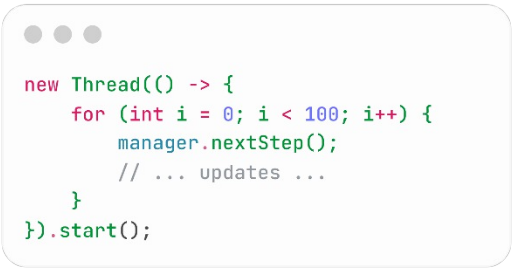
3.1.1. Main Simulation Loop

The core simulation loop is managed by the SimulationManager. While the standard "Step" operation is triggered manually by the user on the Event Dispatch Thread (EDT), the underlying architecture is designed to handle asynchronous updates. The separation of the rendering cycle (MapPanel.repaint()) from the physics calculation (manager.nextStep()) ensures that graphical stuttering does not affect the simulation logic.

3.1.2. Stress Test Implementation

A specific requirement was to verify the system's stability under load. We implemented a "Stress Test" feature that injects a large volume of vehicles (100+) and rapidly advances the simulation.

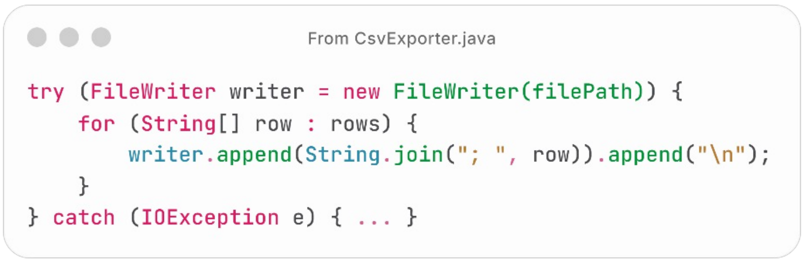
- **Background Thread:** To prevent the GUI from freezing during this intensive process, the stress test logic is wrapped in a separate thread



```
new Thread(() -> {
    for (int i = 0; i < 100; i++) {
        manager.nextStep();
        // ... updates ...
    }
}).start();
```

- **Synchronization:** Since the background thread modifies the vehicle list while the GUI thread might be trying to read it for rendering, we synchronized the critical data access methods. In VehicleRepository, the addVehicle method is marked with the synchronized keyword to prevent race conditions during batch injection.

3.2. File Handling and Streams
The application demonstrates proficient use of Java I/O streams for configuration loading and data persistence.
3.2.1. Configuration Loading
The system initiates by parsing the SUMO configuration file (osm.sumocfg). Additionally, the MapPanel uses ImageIcon and internal streams to load texture resources (vehicle images) from the project directory.
3.2.2. CSV Export Implementation
We implemented a robust two-stage export strategy to optimize performance:
1. **In-Memory Buffer (LogExport):** During the simulation, data (step, time, vehicle count) is collected in a List<String[]>. This avoids the high latency of disk I/O during critical runtime steps.
2. **Batch Write (CsvExporter):** When the export is triggered, the CsvExporter class writes the buffered data to a file. We utilized the **Try-With-Resources** statement to ensure that the FileWriter is automatically closed, preventing resource leaks.


From CsvExporter.java
```java
try (FileWriter writer = new FileWriter(filePath)) {
    for (String[] row : rows) {
        writer.append(String.join("; ", row)).append("\n");
    }
} catch (IOException e) { ... }
```

3.3. Error Handling and Logging
The application implements a comprehensive error handling strategy to ensure robustness against runtime failures (e.g., lost connection to SUMO).
3.3.1. Exception Hierarchy
We handle specific exceptions to provide meaningful feedback:
- **NumberFormatException:** Handled in the GUI (e.g., Traffic Light Duration input) to prevent invalid user data from crashing the app.
- **IOException:** Managed during file export operations.
- **RuntimeException:** Trapped during the main simulation loop to log unexpected SUMO API failures without terminating the session.
3.3.2. Logging Strategy (Log4j 2)
Instead of using standard output (System.out), we integrated the **Log4j 2** library for professional logging. This allows us to categorize messages by severity:
- **logger.info():** Used for major events like "Simulation Started" or "Export Successful".
- **logger.debug():** Used for high-frequency data like step counters.
- **logger.error():** Captures stack traces in catch blocks, aiding in debugging without cluttering the user console.
3.4. Clean Code Principles Applied
The development process adhered to Clean Code standards to ensure readability and maintainability:
- **Separation of Concerns:** The UI logic (MainFrame) is strictly separated from the business logic (SimulationManager).

- **Safe Iteration:** In VehicleRepository, we utilized explicit Iterator loops instead of simple for-each loops when removing vehicles. This prevents ConcurrentModificationException when vehicles leave the simulation boundaries.
- **Meaningful Naming:** Variables and methods use descriptive names (e.g., updateVehicles, switchToNextPhase) rather than abbreviations, making the code self-documenting.

4. Features and Functionalities

This section presents the core capabilities of the application, demonstrating how the technical architecture translates into user-facing features.

4.1. Live SUMO Integration & Map Visualization

The application establishes a real-time TCP connection with the SUMO engine via the TraaS API (SumoTraciConnection). Instead of a static background image, the map is rendered dynamically:

- **Dynamic Road Rendering:** The MapPanel fetches raw lane geometry (SumoGeometry) from SUMO and renders it using Java 2D's Path2D objects. This ensures that the map accurately reflects the simulation environment, regardless of the network topology.
- **Coordinate Transformation:** A custom scaling algorithm translates SUMO's cartesian coordinates (meters) into screen coordinates (pixels).
- **Navigation Controls:** Users can interact with the map using mouse controls:
  - **Zooming:** Mouse wheel to zoom in/out for detailed views of intersections.
  - **Panning:** Click-and-drag to move the camera across the map.

4.2. Vehicle Management (Injection, Grouping, Filtering)

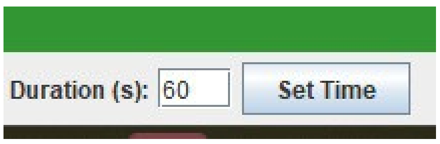
The system offers comprehensive tools for managing traffic flow and analyzing specific vehicle groups.

- **Vehicle Injection:** Users can manually inject vehicles into the simulation using the "Add Car" button. The system ensures unique ID generation (car_1, car_2...) and assigns random start and end edges to create diverse traffic patterns.
- **Stress Testing:** A specialized "Stress Test" mode injects a high volume of vehicles (100+) simultaneously and accelerates the simulation speed. This feature is used to validate the system's performance under heavy load and observe congestion behaviors.
- **Advanced Filtering:** To analyze specific traffic segments, a **Filter Panel** was implemented. Users can filter the visible vehicles on the map based on:
  - **Vehicle Type:** (e.g., Show only "Red" or "Blue" cars).
  - **Street (Edge):** Show only vehicles currently driving on specific roads. This is achieved via the VehicleFilter class, which dynamically updates the MapPanel without affecting the underlying simulation physics.

4.3. Traffic Light Management (Manual & Adaptive Control)

Traffic lights are fully interactive and visualized in real-time.

- **Visualization:** Traffic light states are drawn as colored indicators at the exact stop-line positions calculated from lane geometry.
- **Manual Control:** The "Switch" button allows users to manually force a traffic light to advance to its next phase (e.g., from Red to Green) instantly.
- **Adaptive Duration Control:** Users can modify the phase duration of any selected traffic light dynamically using the "Set Time" input. For example, extending a green phase to 60 seconds to clear congestion.



4.4. Statistics and Real-Time Analytics

To provide immediate feedback on the simulation status, a dedicated **Statistics Panel** is overlayed on the GUI.

- **Live Metrics:** The panel displays the total count of active vehicles in the network.
- **CO2 Emissions Tracking:** The application queries SUMO for the CO2 emission levels of each vehicle, aggregates them, and displays the total emission for the current step.
- **Historical Data:** A history of the last 5 simulation steps is maintained and displayed, allowing users to spot sudden spikes in emissions or traffic density.

4.5. Reporting and Export

For post-simulation analysis, the application includes a data export feature.

- **CSV Generation:** The CsvExporter generates a structured .csv file containing step-by-step data (Simulation Time, Vehicle Count).
- **User-Friendly Saving:** We integrated JFileChooser, allowing users to browse their file system and choose exactly where to save the report (e.g., Desktop or Documents) and what to name it.
- **Data Integrity:** The export process uses buffered data (LogExport) to ensure that even if the simulation is stopped abruptly, the collected data is safely written to the disk.

5. User Manual

This chapter provides a comprehensive guide on how to install, configure, and operate the "Real-Time Traffic Simulation" application.

5.1. Installation and Setup

Before running the application, ensure the following prerequisites are met:

1. **Java Development Kit (JDK):** Ensure JDK 17 or higher is installed.
2. **SUMO Installation:**
   - Download and install the latest version of SUMO (Simulation of Urban MObility) from the official website.
   - **Crucial Step:** Set the environment variable SUMO_HOME to point to your SUMO installation directory (e.g., C:\Program Files\Eclipse\Sumo). The TraaS API requires this to locate the SUMO binaries.
3. **Project Setup:**
   - Import the project into an IDE (Eclipse or IntelliJ).
   - Ensure the SumoConfig folder (containing osm.sumocfg) is present in the project root.
4. **Running the Application:**
   - Navigate to src/trafficsimulation/Main.java.
   - Run the file as a Java Application.

5.2. User Interface Overview

The Graphical User Interface (GUI) is divided into four main functional areas:

1. **The Map Area (Center):** Displays the road network, moving vehicles, and traffic lights.
2. **Statistics Panel (Top-Left):** Shows real-time counters (Vehicle count, CO2 history).
3. **Filter Panel (Right):** Contains checkboxes to filter vehicles by type or street.
4. **Control Panel (Bottom):** Contains all operational buttons (Start, Stop, Add Car, Export, Traffic Light Controls).

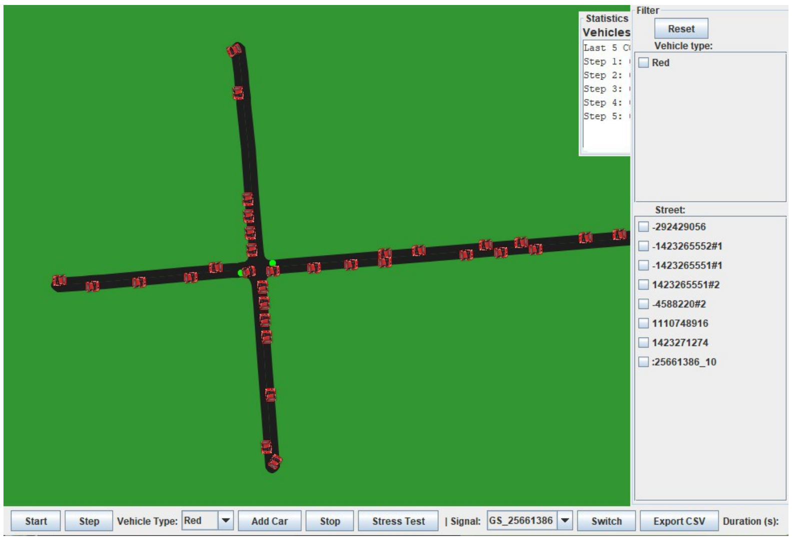


Figure 5.1: The Main Window of the Traffic Simulation Application.

5.3. Step-by-Step Usage Guide

5.3.1. Starting the Simulation

1. Launch the application.
2. Click the **"Start"** button on the bottom control panel.
3. The SUMO GUI window will open in the background, and the application will load the map geometry.
4. Click the **"Step"** button to advance the simulation manually by one time-step, or simply use the controls below to inject traffic.

5.3.2. Adding Vehicles and Stress Testing

There are two ways to add vehicles to the simulation:

- **Manual Injection:**
  1. Select a vehicle color from the dropdown menu (e.g., "Red", "Blue").
  2. Click the **"Add Car"** button. A single vehicle will be inserted at a random valid starting edge.
- **Stress Mode:**
  1. Click the **"Stress Test"** button.
  2. The system will automatically inject **100 vehicles** and advance the simulation rapidly. Use this to observe congestion behavior.

5.3.3. Controlling Traffic Lights

The application allows for both manual switching and duration adjustment of traffic lights:

1. **Select a Junction:** Use the "Signal" dropdown menu to select a specific Traffic Light ID (e.g., cluster_123).
2. **Force Switch:** Click the **"Switch"** button to immediately force the selected light to change to its next phase (e.g., Red → Green).
3. **Change Duration:**
   - Enter a value in seconds (e.g., 45) in the "Duration" text field.
   - Click **"Set Time"**. The new duration will be applied to the current phase in SUMO.

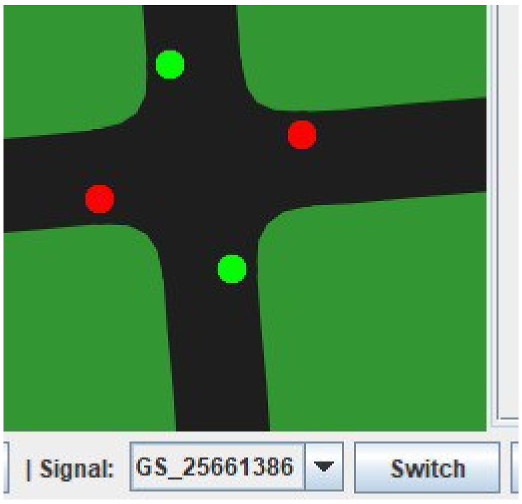


Figure 5.2: Traffic Light Control Interface.

5.3.4. Filtering and Exporting Data

Using Filters: To analyze specific traffic patterns:

1. Look at the **Filter Panel** on the right side.
2. Uncheck specific "Vehicle Types" to hide them from the map.
3. Select specific "Streets" to show only vehicles driving on those edges.
4. Click **"Reset"** to show all vehicles again.

Exporting Reports: To save the simulation data:

1. Click the **"Export CSV"** button.
2. A file chooser dialog will appear. Select the destination folder (e.g., Desktop) and enter a file name (e.g., Run_1.csv).
3. Click "Save". A confirmation message will appear, and the file can be opened in Excel.

6. Project Management and Retrospective

This chapter outlines the organizational aspects of the project, including task distribution, development methodology, and a reflection on the technical challenges encountered.

6.1. Work Distribution (Team Members & Tasks)

The project workload was distributed fairly among all team members, ensuring that everyone contributed to both the "Core Features" and the "Advanced Features". The following table summarizes the primary responsibilities of each member

-Salaheddine Mabchour: Responsible for the system's robustness and the core vehicle data model. He implemented the Error Logging strategy (using Log4j) to track runtime issues. He also developed the VehicleRepository to manage the list of active cars and the VehicleWrap class to encapsulate the TraaS API calls.
-Elias Heß: Focused on the visualization and analytics components. He developed the MapPanel to handle the custom rendering of the road network and vehicles. Additionally, he implemented the StatisticsPanel to display real-time simulation data.
-Yiyuan Ma: Her Task was to manage the traffic light infrastructure and to implemente the TrafficLightRepository and the TrafficLightWrap class, enabling the application to read signal states from SUMO and visualize them on the map.
-Mauricio de Souza Hilpert: In charge of advanced features and quality assurance. He implemented the multi-threaded Stress Test module and the logic for Grouping and Filtering vehicles (e.g., by street or type). He was also responsible for general system testing to ensure stability.
-Laura Gradwohl: Orchestrated the application's control logic and documentation. She implemented the GuiController to link the view with the model, developed the CSV Export

functionality, and took the lead in writing and compiling the final project documentation.
-Shared Responsibilities: The remaining tasks, including the initial architectural setup, merging code branches, and general bug fixing, were handled collaboratively by the entire team.

6.2. Development Methodology

We adopted an **Agile-inspired iterative approach** for the development lifecycle:

1. **Version Control:** We used **Git and GitHub** for source code management. This allowed us to work on different features (Branches) simultaneously and merge them into the main branch after testing.
2. **Milestone Planning:**
    - **Milestone 1:** Focused on establishing the connection with SUMO and rendering the static map.
    - **Milestone 2 (Final):** Focused on dynamic features, user interaction (Stress Test, Filtering), and performance optimization.
3. **Code Reviews:** We held regular meetings to review each other's code, specifically focusing on the separation of concerns (MVC) to ensure the architecture remained clean.

6.3. Challenges and Solutions

During the development process, we faced several technical challenges. Below are the most significant ones and how we resolved them:

Challenge 1: Concurrent Modification Exception

- **Problem:** During the "Stress Test", the simulation thread was adding vehicles to the list while the GUI thread was trying to read it for rendering. This caused the application to crash with ConcurrentModificationException.
- **Solution:** We replaced the standard ArrayList in VehicleRepository with java.util.concurrent.CopyOnWriteArrayList. This thread-safe collection allows concurrent reading and writing without throwing exceptions.

Challenge 2: The "Ghost Car" Issue

- **Problem:** When injecting 100 vehicles at once, SUMO places many of them in a "pending queue" (waiting to enter the map). Our initial logic removed any vehicle not currently "active" on the map, causing these queuing vehicles to be deleted from our Java list before they even appeared.
- **Solution:** We modified the update logic to check the **"Arrived List"** from SUMO. We now only remove a vehicle from our system if SUMO explicitly reports that it has arrived at its destination, ensuring queued vehicles remain tracked.

Challenge 3: Performance Bottlenecks with CSV Export

- **Problem:** Initially, we tried writing to the CSV file at every simulation step. This caused significant lag because disk I/O is slow compared to the simulation speed.
- **Solution:** We implemented an **In-Memory Buffer (LogExport)**. Data is collected in RAM during the simulation and is only written to the disk in a single batch operation when the user clicks "Export".

6.4. Reflection on Team Performance

Overall, the team collaboration was effective. We successfully combined the procedural nature of the TraaS API with a clean Object-Oriented architecture.

- **Strengths:** Good communication and a clear separation of tasks allowed us to integrate complex features like the "Filter Panel" and "Stress Test" without breaking existing functionality.
- **Learning Outcome:** We gained deep insights into **Thread Safety**, **API Wrapping**, and the importance of **Design Patterns (MVC, Observer)** in building robust Java applications. The project not only met the requirements but also provided a practical playground for advanced Java concepts.

6.5. Future Works

Although the application is fully functional, there are several features we would have liked to implement, but did not have enough time for in this group project. These include:

- A more advanced GUI for easier configuration and simulation control which offers more features
- a bigger map to better simulate a whole city
- More advanced validation and error handling to cover edge cases.
- Support for additional SUMO output types and more detailed result analysis.
- Improved documentation and automated tests to increase reliability.

7. Milestones Summary

The project development was divided into two major phases (Milestones) to ensure a structured approach and continuous feedback loop. This section summarizes the objectives and achievements of each phase.

7.1. Milestone 1: Initial Setup and Connectivity

The primary goal of the first milestone was to establish the technical foundation and prove that the Java application could communicate with the SUMO engine.

Key Achievements:

- **Environment Configuration:** Successfully set up the development environment (JDK 17, SUMO Binaries) and integrated the **TraaS (Traffic as a Service)** library into the project build path.
- **Basic Architecture:** Established the skeleton of the **MVC Pattern**. Created the SimulationManager (Model) and the MainFrame (View).
- **SUMO Connectivity:** implemented the SumoTraciConnection logic to launch SUMO and load the configuration file (osm.sumocfg).
- **Static Map Rendering:** Developed the initial version of MapPanel. At this stage, the application could parse road geometry (Lane shapes) and draw the static road network on the screen using Java 2D.
- **Basic Control:** Implemented the "Start" and "Stop" buttons to launch and terminate the SUMO process.

Outcome: A functional "Viewer" application that could start SUMO and display the empty road network.

7.2. Milestone 2: Core Features Implementation

The second milestone focused on interactivity, data management, and fulfilling the advanced grading requirements (Threading, OOP Design).

Key Achievements:

- **Dynamic Entities:** Implemented the VehicleRepository and VehicleWrap classes. Vehicles began to move on the Java map synchronously with the simulation.
- **Interaction & Control:** Added the "Add Car" and "Traffic Light Control" features. Users could now influence the simulation state directly from the GUI.
- **Advanced Features (The "Pro" Version):**
  - **Stress Test:** Implemented the multi-threaded stress test to simulate heavy traffic without freezing the UI.
  - **Filtering:** Added the FilterPanel to allow users to sort vehicles by type or street.
  - **Statistics:** Integrated the live StatisticsPanel for CO2 tracking.
- **Robustness:** Solved critical bugs such as the ConcurrentModificationException (by introducing CopyOnWriteArrayList) and the "Ghost Car" issue (by fixing the update logic).
- **Data Export:** Finalized the CsvExporter for saving simulation logs.

Outcome: The final, fully featured "Real-Time Traffic Simulation" system ready for submission.

Declaration of Authorship

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.

Full Name:	Date:	Signature:
Mabchour Salaheddine	18.01.26	Mabchour Salaheddine
Ma Yiyuan	18.01.26	Ma Yiyuan
Gradwohl Laura	18.01.26	Gradwohl Laura
Mauricio de Souza Hilpert	18.01.26	Mauricio de Souza Hilpert
Elias Heß	18.01.26	Elias Heß

8.2. References and External Libraries

The development of this project relied on the following external software libraries, frameworks, and asset resources:

1. **Eclipse SUMO (Simulation of Urban MObility)**
   - Core simulation engine documentation.
   - Source: Eclipse Foundation (eclipse.org/sumo)
2. **TraaS (Traffic as a Service) API**
   - Java wrapper library for SUMO TraCI.
   - Source: GitHub Repository (TraaS/TraaS)
3. **Apache Log4j 2**
   - Logging framework documentation.
   - Source: Apache Software Foundation
4. **Oracle Java SE 17 Documentation**
   - Standard Java API specification (java.util.concurrent, javax.swing).
   - Source: Oracle Official Documentation
5. **Course Materials**
   - Lecture notes and "Clean Code" guidelines.
   - Lecturer: Prof. Dr.-Eng. Ghadi Mahmoudi (Winter Semester 2025/2026).
6. **PNGWing**
   - Source for graphical assets (Vehicle top-view icons used in MapPanel).
   - Source: pngwing.com

trafficsimulation

Java Dokumentation

Package: trafficsimulation

Anzahl der Klassen: 14

Inhaltsverzeichnis

Klasse Main

Klassendefinition

```
public class Main extends Object
```

Konstruktoren

- Main()

Methoden

- main(String[] args)

Klasse MainFrame

Klassendefinition

```
public class MainFrame extends JFrame
```

Konstruktoren

- MainFrame(MapPanel mapPanel)

Methoden

- getAddCarButton()
- getCarSelector()
- getDurationInput()
- getLightSelector()
- getMapPanel()
- getSetDurationButton()
- getStartButton()
- getStatisticsPanel()
- getStepButton()
- getStopButton()
- getStressTestButton()
- getSwitchLightButton()
- setFilterPanel(FilterPanel filterPanel)

Klasse GuiController

Klassendefinition

```
public class GuiController extends Object
```

Konstruktoren

- GuiController(MainFrame view, SimulationManager manager)

Methoden

- getFilterPanel()
- refreshMap()
- setupFilter(VehicleRepository repo)

Klasse SimulationManager

Klassendefinition

```
public class SimulationManager extends Object
```

Konstruktoren

- SimulationManager()

Methoden

- addListener(trafficsimulation.SimulationListener listener)
- getActiveVehicleCount()
- getConnection()
- getCurrentCo2Emission()
- getLightRepository()
- getRepository()
- getStepCounter()
- nextStep()
- startSimulation()
- stopSimulation()

Klasse VehicleRepository

Klassendefinition

```
public class VehicleRepository extends Object
```

Konstruktoren

- VehicleRepository(it.polito.appeal.traci.SumoTraciConnection conn)

Methoden

- addVehicle(int n, String type, String imageName)
- getAllVehicles()
- getAverageSpeed()
- getList()
- getTotalCo2Emission()
- getvehicleCounter()
- updateVehicles()

Klasse VehicleWrap

Klassendefinition

```
public class VehicleWrap extends Object
```

Konstruktoren

- VehicleWrap(String id)
- VehicleWrap(String id, it.polito.appeal.traci.SumoTraciConnection conn, String imageName)

Methoden

- getAngle()
- getCo2Emission()
- getColor()
- getEdge()
- getId()
- getID()
- getImageName()
- getPosition()
- getSpeed()
- getX()
- getY()
- setEdge(String edge)
- updateVehicle()

Klasse VehicleFilter

Klassendefinition

```
public class VehicleFilter extends Object
```

Konstruktoren

- VehicleFilter(VehicleRepository repo)

Methoden

- addEdge(String edge)
- addType(String type)
- clear()
- getAllEdges()
- getAllTypes()
- getFiltered()
- removeEdge(String edge)
- removeType(String type)

Klasse TrafficLightRepository

Klassendefinition

```
public class TrafficLightRepository extends Object
```

Konstruktoren

- TrafficLightRepository(it.polito.appeal.traci.SumoTraciConnection conn)

Methoden

- findlight(String id)
- getList()
- loadLights()
- updateLights()

Klasse TrafficLightWrap

Klassendefinition

```
public class TrafficLightWrap extends Object
```

Konstruktoren

- TrafficLightWrap(String id, it.polito.appeal.traci.SumoTraciConnection conn)

Methoden

- getCo2Emission()
- getCurrentState()
- getID()
- getSignalPoints()
- setPhaseDuration(double seconds)
- switchToNextPhase()
- updateState()

Klasse MapPanel

Klassendefinition

```
public class MapPanel extends JPanel
```

Konstruktoren

- MapPanel(SimulationManager manager)

Methoden

- centerMap()
- loadMap()
- paintComponent(Graphics g)
- updateVehicles(List vehicles)

Klasse FilterPanel

Klassendefinition

```
    public class FilterPanel extends JPanel
```

Konstruktoren

- FilterPanel(VehicleFilter filter, GuiController controller)

Methoden

- update()

Klasse StatisticsPanel

Klassendefinition

```
public class StatisticsPanel extends JPanel
```

Konstruktoren

- StatisticsPanel()

Methoden

- addCo2Value(double value)
- setVehicleCount(int count)

Klasse CsvExporter

Klassendefinition

```
public class CsvExporter extends Object
```

Konstruktoren

- CsvExporter()

Methoden

- export(String filePath, List rows)

Klasse LogExport

Klassendefinition

```
public class LogExport extends Object
```

Konstruktoren

- LogExport()

Methoden

- getRows()
- logStep(int step, double time, int vehicleCounter)