Practical Assignment 1

Multithreaded City Traffic

1 Assignment Overview

Modern cities increasingly rely on autonomous and connected vehicles to improve traffic efficiency, safety, and energy sustainability. At the same time, the correct coordination of multiple independent agents competing for shared resources is a central challenge in **concurrent and distributed systems**. This practical work explores these challenges through the simulation of a **multithreaded city traffic** system, implemented in **Java**.

In this assignment, the city is modeled as a two-dimensional grid, where each cell represents a road intersection. Autonomous electric vehicles circulate through this city, moving from one intersection to another over a predefined number of cycles. Each vehicle operates independently and is represented by a dedicated thread, executing concurrently with all others. As vehicles move through the city, they must coordinate their access to intersections in order to avoid collisions, respecting traffic constraints such as turning rules and mutual exclusion. For instance, when a vehicle performs a u-turn, it must have exclusive access to the intersection, whereas in other situations multiple non-conflicting vehicles may cross simultaneously.

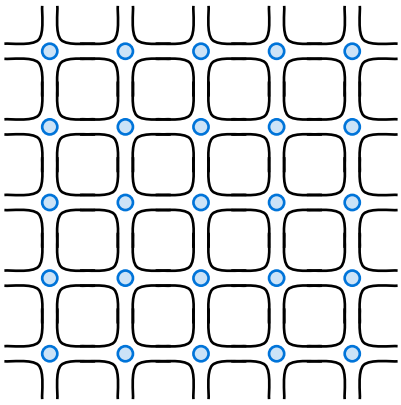

Figure 1: Example of a city grid with size 5×5. Each blue dot marks an intersection.

Beyond traffic coordination, the system also models resource constraints related to energy management. Each vehicle has a limited battery capacity that is depleted as it moves through the city. When the energy level falls below a defined threshold, the vehicle must interrupt its current route and autonomously navigate to the nearest charging station. Charging stations are shared resources with a limited number of charging spots and a bounded power capacity, requiring vehicles to coordinate access and possibly wait until resources become available. Once recharged, vehicles resume their original objectives.

The goal of this practical work is to design and implement a correct, and well-structured concurrent solution using Java's concurrency mechanisms. Students are expected to reason about synchronization, mutual exclusion, condition synchronization, and fairness, while avoiding common concurrency pitfalls such as race conditions, deadlocks, and starvation. The resulting system should faithfully simulate realistic traffic behavior while serving as a concrete case study of concurrency control in a complex, shared-resource environment.

2 Objectives and Requirements

The primary objective of this practical assignment is to design and implement a concurrent simulation of an urban traffic system, using **Java multithreading** as the core execution model. Each group must achieve the following objectives:

1) **Model the lifecycle of concurrent entities**
   Design clear and well-defined lifecycles for all active elements of the simulation, including vehicles and system controllers. Each autonomous vehicle must be represented by an independent thread whose behavior evolves over time according to the simulation rules (movement, waiting, charging, resuming routes).

2) **Apply correct synchronization mechanisms**
   Use Java's concurrency primitives to ensure safe and consistent access to shared resources such as road intersections and charging stations. Students are expected to correctly apply mutual exclusion, condition synchronization, and signaling, following the principles of concurrent programming introduced in the course.

   > ✗ **Forbidden**
   >
   > The use of Java's `synchronize` keywork is not allowed. You must use concurrency mechanisms available at `java.util.concurrent`

3) **Reason about concurrency correctness and fairness**
   Ensure that the system is free from race conditions, deadlocks, livelocks, and starvation. The solution should exhibit fair access to shared resources whenever applicable, particularly under high contention.

4) **Develop a graphical user interface (GUI)** (optional for groups of 2)
   Implement a GUI that visually represents the current state of the simulation, enabling intuitive observation of vehicle movement, intersection usage, and charging station occupancy. The GUI can also support basic control over the simulation execution.

2.1 Core Simulation Requirements

The simulation environment must adhere to the following structural and behavioral specifications to ensure a consistent and thread-safe distributed environment:

2.1.a Topology and Infrastructure

- The city is represented as a discrete two-dimensional grid of $R \times C$ dimensions.
- Each cell in the grid is an `Intersection` object, which acts as a shared resource for vehicles.
- An `Intersection` may optionally host a `ChargingStation`, providing battery charging capabilities.
- The number of `ChargingStation` instances deployed in the grid must not exceed 25% of the total number of intersections.
- Movement between intersections is restricted to cardinal directions: `North`, `East`, `South`, and `West`.
- A single conceptual lane exists for each direction between adjacent intersections, necessitating coordination to prevent collisions or deadlocks.

2.1.b Vehicle Dynamics and Lifecycle

- Each `Vehicle` is implemented as an independent Java thread.

- Vehicles are initialized with a random starting `Intersection` and `Direction`, and a maximum number of simulation steps.
- A vehicle's execution loop continues until it either completes its assigned steps or its battery level reaches zero.
- Every move between intersections consumes battery power, requiring vehicles to navigate toward charging stations when energy is low.
- Vehicles must follow a strict "Enter-Traverse-Exit" protocol when interacting with intersections to ensure mutual exclusion or limited occupancy.

2.1.c Simulation Constraints and Timing

- The simulation's temporal resolution is defined by a configurable time unit (ms).
- All discrete movements and charging actions must be scaled according to this user-defined duration, with a minimum value of millisecond.
- The simulation must support a configurable number of vehicles, rows, and columns through command-line arguments, allowing for scalability testing.
- The main orchestration thread must wait for the termination of all vehicle threads before finalizing the simulation.

2.2 Vehicle Lifecycle and Autonomous Behavior

Each vehicle in the simulation acts as an independent, autonomous agent responsible for its own pathfinding, energy management, and resource coordination. This section details the operational logic that every `Vehicle` thread must implement.

2.2.a Active Entity Model

Every vehicle is an **active entity** implemented as a dedicated Java thread. The vehicle's lifecycle consists of a continuous sequence of movement cycles. A cycle begins when a destination is selected and ends when the vehicle successfully arrives at that location.

2.2.b Energy and Power Model

Vehicles are constrained by finite energy resources, necessitating strategic movement and interaction with charging infrastructure.

Parameter	**Requirement**
Initial Battery	All vehicles start the simulation with a full charge of 100%.
Power Requirement	At initialization, each vehicle is assigned a constant charging power requirement between 50 and 100 power units (uniformly distributed). This represents the load the vehicle places on a `ChargingStation` during a recharge.
Consumption	To traverse an intersection a vehicle must consume between 1% and 2% of its battery. To travel from one intersection to another the a vehicle must consumme between 2% an 5% of its battery. Theses values are drawn from a uniform distribution.
Termination	If a vehicle's battery level reaches 0%, the thread must terminate its execution at the end of the step.

2.2.c Operational Timings

To simulate physical movement and delays, vehicles must yield execution (sleep) based on the following temporal constraints:

- **Inter-Intersection Travel**: Moving from one intersection to the next takes a random duration between 10 and 50 time units.
- **Intersection Traversal**: The process of physically crossing an intersection (once access is granted) takes between 5 and 25 time units.

2.2.d Autonomous Navigation and Cycles

Vehicles must exhibit autonomous decision-making through the following loop:

- **Destination Selection**: Choose a random `Intersection` within the `CityMap` as the target.
- **Pathfinding**: Determine a route from the current coordinate to the destination.
- **Execution**: Traverse the grid one intersection at a time.
- **Recharging Diversion**: If the battery falls below a specific threshold, the vehicle must temporarily override its current objective to navigate to the nearest `ChargingStation`. After recharging, it must resume its original goal.
- **Cycle Completion**: Upon reaching the destination, the vehicle immediately begins a new cycle.

2.3 Intersection Coordination and Traffic Control

Intersections are the primary shared resources in the city grid and must be protected against unsafe concurrent access to prevent collisions and deadlocks. Students must implement a robust coordination layer within the `Intersection` class to manage vehicle flow according to specific traffic rules.

2.3.a Navigation Semantics

The simulation distinguishes between the physical orientation of the vehicle and its intended behavior within the intersection:

- **Entry and Exit Points (`Direction`)**: The `NORTH`, `SOUTH`, `EAST`, and `WEST` constants define the physical entry points into the intersection and the subsequent exit points after traversal.
- **Internal Motion (`MoveType`)**: The `MoveType` enum (`STRAIGHT`, `LEFT_TURN`, `RIGHT_TURN`, `U_TURN`) defines the trajectory a vehicle follows once it has been granted access to the intersection.

2.3.b Intersection Occupancy Rules

To maximize throughput while ensuring safety, the intersection utilizes a fine-grained occupancy model based on a collision matrix. Unlike a simple exclusive-access model, multiple vehicles may occupy the intersection simultaneously provided their intended trajectories do not intersect.

2.3.b.a Conflict-Based Concurrent Access

- **Mutual Exclusion**: A vehicle is permitted to enter the intersection only if no other vehicle currently traversing the intersection is performing a movement that collides with its own.
- **Directional Symmetry**: Although the collision matrix is defined relative to the `NORTH` entry point, the same logic applies symmetrically to `SOUTH`, `EAST`, and `WEST` entries (e.g., follows the same conflict patterns as relative to its own oncoming and side neighbors).
- **State Management**: The synchronization logic must track the `Direction` and `MoveType` of every vehicle currently inside the intersection to evaluate these safety constraints in real-time.

Paths are denoted as X_Y, where X is the Entry Direction (N, S, E, W) and Y is the Move Type (S: Straight, L: Left, R: Right, U: U-Turn). For example, E_S represents a vehicle entering from the East and moving Straight.

The following table defines the specific conflicts for a vehicle entering from the **North**:

North	vs. South (Oncoming)	vs. East (Right Side)	vs. West (Left Side)
`STRAIGHT`	Collides: S_L, S_U	Collides: E_S, E_L, E_U	Collides: `all`
`LEFT_TURN`	Collides: `all`	Collides: E_S, E_L, E_U	Collides: `all`
`RIGHT_TURN`	Collides: S_L, S_U	Collides: E_S, E_U	Collides: W_U
`U_TURN`	Collides: `all`	Collides: `all`	Collides: `all`

2.3.b.b Movement Implications

- **Right Turns**: These are considered universally safe relative to other entry points (assuming merge-only behavior) and can generally proceed without waiting for the intersection to clear, provided the entry point lock is acquired.
- **Straight and Left Turns**: These movements have high conflict rates with side-street traffic (`EAST` and `WEST`) and oncoming traffic, requiring significant coordination.
- **U-Turns**: These are the most restrictive movements, colliding with nearly all other traffic flows except for non-conflicting right turns from specific directions.

2.3.c Interaction Protocol: Enter and Exit

Vehicles must strictly adhere to the "Enter-Traverse-Exit" protocol by calling the provided methods in the `Intersection` class. These methods serve as the synchronization barriers.

2.3.c.a The `enter()` Method

Vehicles must call `enter(Direction entry, MoveType move, String vehicleId)` before beginning any movement into the intersection.

- **Blocking Behavior**: This function must block the calling thread until the requested entry point is free AND the intersection is available according to the occupancy rules (e.g., waiting for a `U_TURN` to clear).
- **Resource Acquisition**: Once the thread resumes, it signifies that the vehicle has "acquired" the intersection. The implementation must increment the `entryCount` and log the acquisition event as specified in the template.

2.3.c.b The `exit()` Method

Upon completing the traversal (after the simulated delay), the vehicle must call `exit(Direction entry, MoveType move, String vehicleId)`.

- **Resource Release**: This function is responsible for releasing all held synchronization primitives (entry point locks, intersection occupancy permits).
- **Notification**: The implementation must ensure that waiting vehicles are notified that resources have become available.
- **State Update**: The `entryCount` must be decremented, and the final exit log must be generated.

2.3.d Simulation Logging

To ensure traceability, vehicles must log their state transitions. This includes entering/exiting intersections (`>|`, `><`, `>>`, `<<` markers) and reporting battery levels at the Start of Step (SoS) and End of Step (EoS) as indicated in the provided `Vehicle.java` template.

2.4 Energy Management and Charging Infrastructure

Energy management is a critical component of the autonomous vehicle lifecycle. Vehicles must monitor their battery levels and prioritize recharging to prevent system-level failure (0% battery).

2.4.a Battery Constraints and Recharging Logic

Each vehicle has a finite battery capacity (100%) that decreases with every intersection traversal.

- **Threshold Monitoring**: When a vehicle's battery level drops below a predefined (or dynamic) safety threshold (e.g., 20%), it must immediately interrupt its current destination.
- **Autonomous Redirection**: The vehicle must use the `CityMap` to identify and navigate to the nearest intersection equipped with a `ChargingStation`.
- **Mission Resumption**: Once the battery is replenished to 100%, the vehicle must calculate a new path to its original destination.

2.4.b Charging Station as a Shared Resource

Charging stations are concurrent resources that impose both physical and electrical constraints on the simulation.

- **Plug Constraints**: Each station has a limited number of charging plugs (represented by `availablePlugs`). The total number of plugs per charger should be approximately $\frac{1}{4}$ the number of active vehicles to ensure resource contention. The minimum is 1 plug per charger.
- **Power Budget**: Stations have a bounded `availablePower` capacity. This capacity is calculated as $100 + (\text{numPlugs} - 1) \times 65 \pm 15$, where `numPlugs` is the number of plugs at that station.
- **Concurrent Access**: A vehicle may only begin charging if:
  1) There is at least one **available plug**.
  2) The station's **available power** is greater than or equal to the vehicle's specific **charging power requirement** (assigned at the beginning of the simulation).

2.4.c The Charging Protocol (`useCharger`)

The `useCharger` method in the `ChargingStation` class serves as the synchronization point for energy replenishment.

Phase	**Requirement**
Resource Wait	If no plugs are free or the power budget is exceeded, the vehicle thread must block (wait) until resources are released by other vehicles.
Acquisition	Upon entry, the vehicle must decrement the `availablePlugs` and subtract its power requirement from the station's `availablePower`.
Simulation	Charging is simulated by a time between 10 and 50 units of time (uniformly distributed).
Release	After charging, the vehicle must release the plug and return the power to the station's budget, notifying any waiting threads.

2.5 Graphical User Interface (GUI)

The simulation includes a graphical component to provide real-time visualization of the city grid, vehicle movements, and charging station occupancy.

2.5.a Requirements and Optionality

- **Group Size Context**: The implementation of the GUI is **optional for groups of two (2) students**. For these groups, a fully functional GUI may serve as a basis for the "Innovation" or "Solution Adequacy" evaluation criteria.
- **State Visualization**: The GUI must represent the current state of the `CityMap`, including:
  - The location and direction of all active `Vehicle` threads.
  - The status of `ChargingStation` resources (available plugs and power levels).
  - Visual indicators for vehicles currently traversing an `Intersection`.

2.5.b Implementation Constraints and Decoupling

To maintain a clean architectural separation between the simulation logic and the presentation layer, the following constraints must be respected:

- **Separation of Concerns**: GUI logic must be strictly isolated from the simulation entities. Entities participating in the simulation—specifically the `Vehicle`, `Intersection`, and `ChargingStation` classes—must not contain any references to GUI components or UI-specific library calls.
- **State Observation**: The GUI should observe the state of the simulation (e.g., by periodically polling the `CityMap` or using a thread-safe observation mechanism) rather than having the entities "push" updates directly to UI elements.
- **Provided Template**: Students should utilize or extend the `SimulationGUI` class located in the `deti.sd.mt.ct.ui` package. The `Simulation` main class already includes a `--gui` (or `-g`) command-line flag to trigger the interface.

2.5.c Performance and Concurrency

- The GUI must run in its own execution thread (typically the JavaFX or Swing Event Dispatch Thread) to ensure that interface rendering does not block the progress of the vehicle threads.
- Access to shared data structures for rendering must be performed in a thread-safe manner, ensuring that the GUI displays a consistent snapshot of the simulation without interfering with the active `Vehicle` operations.

2.6 Extra Tasks

Groups are encouraged to implement additional features to further explore concurrency and distributed systems challenges. These enhancements contribute to the "Innovation" score. Possible tasks include:

- **Priority-Based Traffic**: Implementing vehicles with different priority levels (e.g., emergency vehicles) that must be granted preferential access to intersections and charging plugs.
- **Advanced Topology**: Expanding the city grid to support two lanes per direction, requiring more complex coordination logic to manage lane changes and parallel traversal.
- **Demand-Responsive Service**: Transforming the simulation into an "Uber-like" system where vehicles navigate to specific coordinates based on dynamic passenger pick-up and drop-off requests.

3 Architectural Constraints and Provided Base Code

All groups **must adhere to the proposed reference architecture** provided with this assignment. To ensure consistency, comparability of results, and a common baseline for evaluation, a **base code skeleton** implementing this architecture will be supplied to each group at the start of the project.

The provided architecture defines:

- The **main structural components** of the simulation (e.g., simulation core, active entities, shared resources, and visualization layer).
- The **execution model**, including thread creation, lifecycle management, and termination.
- A fixed set of **execution options** (e.g., command-line arguments or configuration parameters) used to launch and control the simulation.
- A set of **mandatory outputs**, such as console logs, status reports, or structured data produced during or at the end of the simulation.

Groups are **allowed to extend, refactor, or optimize** the proposed architecture, including:

- Introducing new classes or interfaces.
- Reorganizing internal components.
- Replacing internal synchronization strategies.

However, the following constraints are **mandatory**:

- **Execution compatibility**: All defined execution options must be preserved exactly as specified in the base code. The simulation must be runnable using the same commands and parameters as the reference implementation.
- **Output compatibility**: Any required output format, semantics, and content defined in the base code must be maintained. This ensures that automated testing and evaluation tools can be applied uniformly across all submissions.
- **Architectural compliance**: The overall architectural roles (active entities vs. passive shared resources, simulation controller, GUI integration) must remain identifiable and consistent with the reference design, even if their internal implementation changes.
- **Behavioral equivalence**: Any architectural modification must preserve the observable behavior of the system with respect to concurrency, synchronization constraints, and simulation rules.

Failure to comply with the provided architecture, execution options, or required outputs may result in the solution being **non-evaluable** or **penalized**, regardless of the internal quality of the implementation.

4 Evaluation

The evaluation of this project is divided into two main components: the technical implementation (90%) and a final presentation (10%). **The presentation is mandatory**. All groups must maintain their source code in a dedicated GitHub repository, which must include a `README.md` file documenting the architectural decisions and concurrency strategies employed.

4.1 Presentation (10%)

Following the code submission, each group will perform a presentation. The total duration is strictly limited to **15 minutes**, structured as follows:

- **Presentation (5 min):** A concise overview of the synchronization mechanisms and logic used to prevent deadlocks and starvation.

- **Demo & Q&A (10 min):** A live demonstration of the software, during which the instructor may ask technical questions regarding the implementation.

4.2 Technical Implementation (90%)

The submitted project will be evaluated based on the following weighted criteria, focusing on technical correctness, software engineering standards, and documentation quality.

Evaluation Criteria

- **Functionality (55%)**
  - All core features are working correctly (45%)
    - with 10 successful, error-free runs
  - No deadlock or starvation observed (10%)
- **Code Quality (15%)**
  - Proper use of object-oriented programming and modularity (10%)
  - Code readability and comments (5%)
- **Documentation (15%)**
  - Complete and clear README (10%)
    - Instructions on how to compile and run the application.
    - An explanation of the synchronization strategy used in the different entities.
    - How the group specifically addressed the prevention of **Deadlock** and **Starvation**.
    - Other information considered relevant by the group.
  - Technical documentation quality (5%)
- **Innovation (15%)**
  - Implementation of optional features (10%)
    - Only if all core features are working correctly
  - Creative solutions to problems
  - User experience enhancements
- **No GUI Implementation (−15%)**
  - **Only for groups of three students**
    - failure to implement a GUI will result in a 15% penalty.
    - A well-designed and fully implemented GUI incurs no penalty.

4.3 Academic Integrity Guidelines

This assignment is a **collaborative group effort**. While you are encouraged to discuss general concepts with peers from other teams, your final codebase must be the **unique, original creation** of your specific group.

Usage of AI & External Tools

- **Permitted:** Using AI assistants to troubleshoot bugs or clarify complex theoretical concepts.
- **Prohibited:** Using AI to generate entire solutions or major functional blocks of code.
- **Citations:** You must explicitly credit any external code snippets or concepts sourced from the web or elsewhere.

Standards & Consequences

- **Formatting:** All references and citations must strictly adhere to **IEEE templates**.
- **Plagiarism:** There is a zero-tolerance policy for academic dishonesty.
- **Penalty:** Any group found in violation of these rules will face **immediate disqualification** of their work.

5 Delivery

To ensure a smooth evaluation process, students must adhere to the following delivery requirements. The state of the repository at the deadline will be considered the final submission for grading.

5.1 Submission Requirements

The project delivery consists of two mandatory components hosted on GitHub:

- **Source Code:** A fully functional Java project.
- **Documentation:** A `README.md` file located in the root directory.

5.2 Deadline and Protocol

The submission window closes on **March 25 at 23:59**.

- **Automated Retrieval:** At the deadline, the teaching staff will pull the code directly from the main branch of your GitHub repository.
- **Late Submissions:** Any delays must be **explicitly declared** by the group via email before the deadline.
- **Version Control:** We strongly recommend frequent commits. The commit history will be used to verify the group's contribution distribution and the evolution of the solution.

> ⚠ **Warning**
>
> Code pushed after the deadline without a prior, approved declaration will not be considered for evaluation. Ensure your repository is public or that the instructor has been granted access prior to the cutoff.