Group Project

Phase 1

DM857 Introduction to Programming
DS830 Introduction to Programming

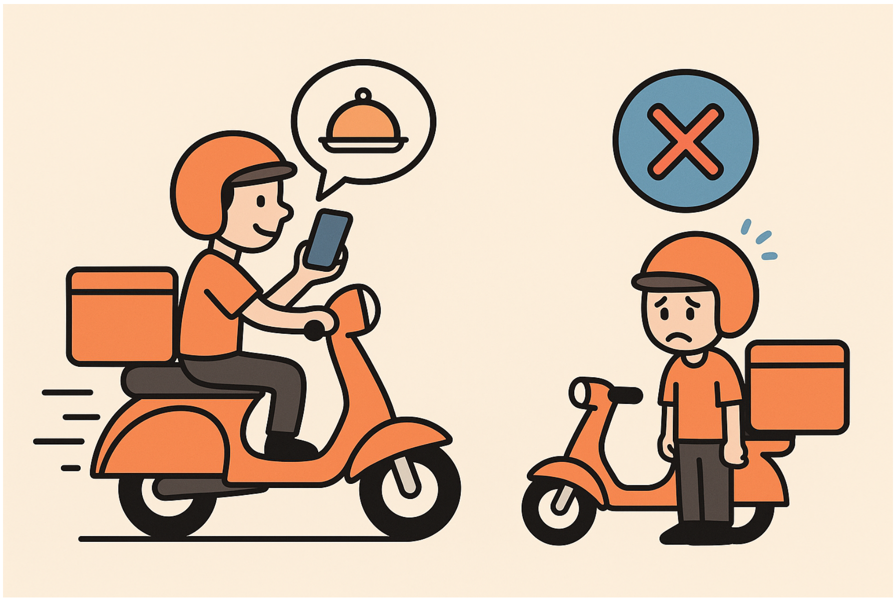
A Food Delivery Service

This document describes **Part I of the annual programming project** for the courses DS830 and DM857: Introduktion til Programmering.
The overall objective of the project is to design and implement the logic of a **simulation representing a food delivery system**, similar in spirit to platforms such as Uber Eats or Wolt.
In this first phase, we focus on a simplified model involving two main groups of elements:

- **Drivers**, representing the agents performing deliveries.
- **Requests**, representing food orders appearing over time.

Most initial parameters of the simulation will be provided through a dedicated graphical interface prepared for the course, which will be described later in the document.
In this first part, we describe the simulation logic, data structures, and required student functions, assuming that once implemented, these will be integrated into the graphical interface and receive initialization parameters automatically.

1 The Simulation



The simulation takes place in a flat, rectangular space with no obstacles. Drivers receive delivery orders defined by a pickup location and a delivery destination. Each order has an expiration time, after which it is no longer available. The simulation assigns drivers to orders according to a chosen logic, and the drivers move within the space—first to the pickup point, then to the delivery point—before becoming available again for new assignments.

2 Simplified Model

Drivers: The number of drivers is fixed at initialization time. Each driver is characterized:

- a unique identifier `id`;
- a spatial position `(x, y)` on the grid;
- a velocity vector `(vx, vy)` and a typical speed `speed`;
- a reference to a target request through its `target_id`, if any.

Requests: Requests appear dynamically according to a chosen frequency. Each request is described by:

- a unique identifier `id`;
- pickup coordinates `(px, py)` and delivery coordinates `(dx, dy)`;
- the time `t` at which the request appeared;
- the identifier of the driver currently associated (if any);
- a |`status`| flag describing its state in the delivery process:
  |`"waiting" | "assigned" | "picked" | "delivered" | "expired"`|.

3 Project Aim

The purpose of this first part of the project is to build a **minimal yet complete simulation engine** supporting this logic. Students are required to implement exactly **six functions with fixed interfaces**, which together form the *procedural backend* used by the visualization system.

These functions provide the basic mechanics to:

- load or generate driver and request data;
- initialize the simulation state;
- update it step by step in time, handling movement, matching, and delivery completion.

Once these six functions are implemented, they can be connected to the graphical user interface provided in `dispatch_ui.py`, which visualizes the evolving system in real time.

4 Procedural Backend Specification

The following sections define precisely the six functions that compose the backend API expected by the user interface.

The UI expects a backend mapping with the following keys:

$$\texttt{BackendFns} = \{\ \text{load\_drivers},\ \text{load\_requests},\ \text{generate\_drivers},\ \text{generate\_requests},\ \text{init\_state},\ \text{simulate\_step}\ \}$$

Each entry is a callable with the signature and semantics defined below.

4.1 Coordinate, Time, and Status Conventions

- **Grid bounds:** $x \in [0, \texttt{GRID\_WIDTH}]$, $y \in [0, \texttt{GRID\_HEIGHT}]$ (default 50×30).
- **Time unit:** minutes (integers). Simulation clock: `state["t"]`.
- **Request lifecycle:**
  |`"waiting"`| $\rightarrow$ |`"assigned"`| $\rightarrow$ |`"picked"`| $\rightarrow$ |`"delivered"`| (or |`"expired"`| if the timeout is exceeded).

4.2 Data Schemas

All data are encoded as plain Python dictionaries with the following keys. Note that many arguments are described as optional, it is left to the student to choose the most appropriate parameters for their simulation routines.

4.2.1 Driver

```
driver = {
  "id": int|str,
  "x": float, "y": float,
  "vx": float, "vy": float,     # velocity (optional)
  "speed": float,               # speed (optional)
  "tx": float, "ty": float,     # target position (optional)
  "target_id": int|str          # optional: id of assigned request
}
```

4.2.2 Request

```
request = {
  "id": int|str,
  "px": float, "py": float,     # pickup
  "dx": float, "dy": float,     # dropoff
  "t": int,                     # appearance time
  "t_wait": int,                # waiting time (optional)
  "status": "waiting" | "assigned" | "picked" | "delivered" | "expired",
  "driver_id": int|str | None
}
```

4.2.3 Simulation State

```
state = {
  "t": int,
  "drivers": list[driver],
  "pending": list[request],
  "future": list[request], # Optional: handles requests that are scheduled for simulation steps c
  "served": int,
  "expired": int,
  "timeout": int,
  "served_waits": list[float],
  "req_rate": float,
  "width": int, "height": int
}
```

4.2.4 Metrics

```
metrics = {
  "served": int,
  "expired": int,
  "avg_wait": float
}
```

5 Backend Function Specifications

5.1 `load_drivers(path) -> list[dict]`

Load driver records from a file and return a list of dictionaries with positions and optional velocity or target information. The request is not to use any external libraries, but rather to write your own parsers and validators for the CSV file.

5.2 `load_requests(path) -> list[dict]`

Load request records from a file and return a list of request dictionaries initialized with defaults for missing fields. The request is not to use any external libraries, but rather to write your own parsers and validators for the CSV file.

5.3 `generate_drivers(n, width, height) -> list[dict]`

enerate n random drivers uniformly within the grid bounds.

5.4 `generate_requests(start_t, out_list, req_rate, width, height) -> None`

Append new requests to `out_list` based on a generation rate (requests per minute).

5.5 `init_state(drivers, requests, timeout, req_rate, width, height) -> dict`

Initialize and return the full simulation state dictionary containing counters, the grid size (fixed for phase 1 of the project), and all entities. The input requests must be generated from the output of the `generate/load` functions defined above.

5.6 `simulate_step(state) -> (state, metrics)`

Advance the simulation by one discrete step. Handle arrivals, assignments, movements, and delivery or expiration events, returning the updated state and metrics.

This is the core of the simulation: it must take the input state and evolve it by one time step (one unit of time). This involves describing how

1. drivers move depending on their status and their speed,
2. whether they manage to pick up or deliver a parcel,
3. how new orders are generated, and
4. how standing orders are assigned to waiting drivers.

6 Minimal Backend Skeleton

```python
def load_drivers(path): ...
def load_requests(path): ...
def generate_drivers(n, w, h): ...
def generate_requests(start_t, out_list, rate, w, h): ...
def init_state(drivers, requests, timeout, rate, w, h): ...
def simulate_step(state): ...
```

Once implemented, group them as:

```python
backend = {
  "load_drivers": load_drivers,
  "load_requests": load_requests,
  "generate_drivers": generate_drivers,
  "generate_requests": generate_requests,
  "init_state": init_state,
  "simulate_step": simulate_step,
}
run_app(backend)
```

7 Post-Simulation Metrics and Reporting

After closing the graphical application window, the program should display a new window or generate plots summarizing the **evolution of the simulation metrics** over time. This secondary output is intended to allow inspection and analysis of how the system performed during the simulation (for example, trends in served or expired requests, or the evolution of average waiting times).

7.1 Requirements and design

The exact implementation of this feature is left to the students. The provided graphical interface (`dispatch_ui.py`) does not define any specific output channel or data format for this post-simulation report. Students must therefore design and implement an appropriate method to:

- store the relevant metrics at each simulation step during execution;
- retrieve these data after the application window is closed;
- present them in a clear and informative way, for example by generating a matplotlib plot.

The goal of this additional step is to demonstrate that your backend correctly maintains and exposes simulation statistics, and that you can design a simple reporting interface based on your own data management choices.

8 Hand-in

8.1 Files

For this phase, submit a zip file containing:

- A folder named code with your implementation, including a module named `phase1`.
- A PDF document titled `named_report.pdf` containing your report. The zip file should be named after your group, e.g., `group_B40.zip`.

8.2 Setup

- Your solution must contain a module named `phase1`, which serves as the main module of your program; otherwise, you may organize your solution as you see fit.
- Do not modify any modules provided with this assignment (`_engine.py`).
- Document your code, adhering to the Python coding conventions and course rules.
- Use only modules from the Python standard library, plus `numpy`, `matplotlib`, `os`,and `dearpygui` (other third-party packages are not allowed).
- In your report, describe your implementation, explaining and justifying its structure and functionality.
- Include your code[1] as an appendix. To save space, you may summarize or omit docstrings, doctests, and code for printing configurations and statistics.
- No specific report structure is required beyond including a code appendix.
- The report must be in English and delivered as a single PDF, printable in black and white (avoid dark backgrounds for code listings) and limited to 10 pages, excluding the appendix.
- The report must state the group's name and list its members (full names and SDU emails, if applicable).

[1] If preparing your report in LaTeX, use packages such as "minted" or "listings" for source code.