1 Aufgabenstellung

1.1 Aufgabe: Entwicklung eines Dashboards für mein Studium

1.1.1 Konzeptionsphase

1. Festlegung der zu überwachenden Ziele

Im Dashboard sollen drei Kernaspekte kontinuierlich visualisiert werden:

a) **Zeitplan** – Wie gut halte ich den vorgegebenen Studienzeitplan ein? Das System zeigt, ob ich mich im Plan befinde und empfiehlt bei Abweichungen konkrete Maßnahmen, um wieder auf Kurs zu kommen.

b) **Notenentwicklung** – Ziel ist ein Abschluss mit einem Durchschnitt von 2,3 oder besser. Das Dashboard stellt den aktuellen Schnitt dar und macht Trends sofort sichtbar.

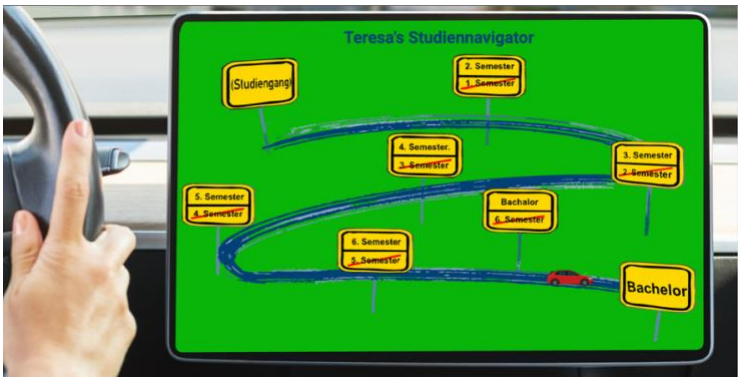
c) **Finanzstatus** – Wie viel der Studiengebühren ist bereits beglichen und welcher Restbetrag steht noch aus? So behalte ich meine finanziellen Verpflichtungen stets im Blick.

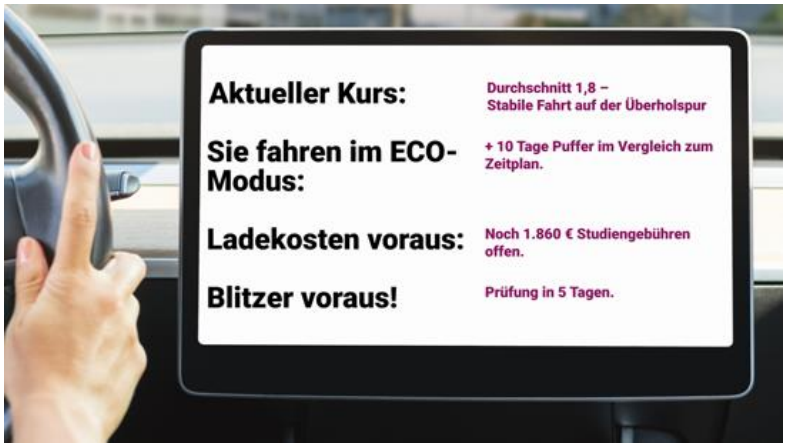
2. Festlegung der anzuzeigenden Daten und Visualisierung

Um diese Kennzahlen anschaulich darzustellen, setze ich auf ein **Dashboard in Form eines Navigationssystems** – inspiriert durch meine Arbeit in der Automobilindustrie.

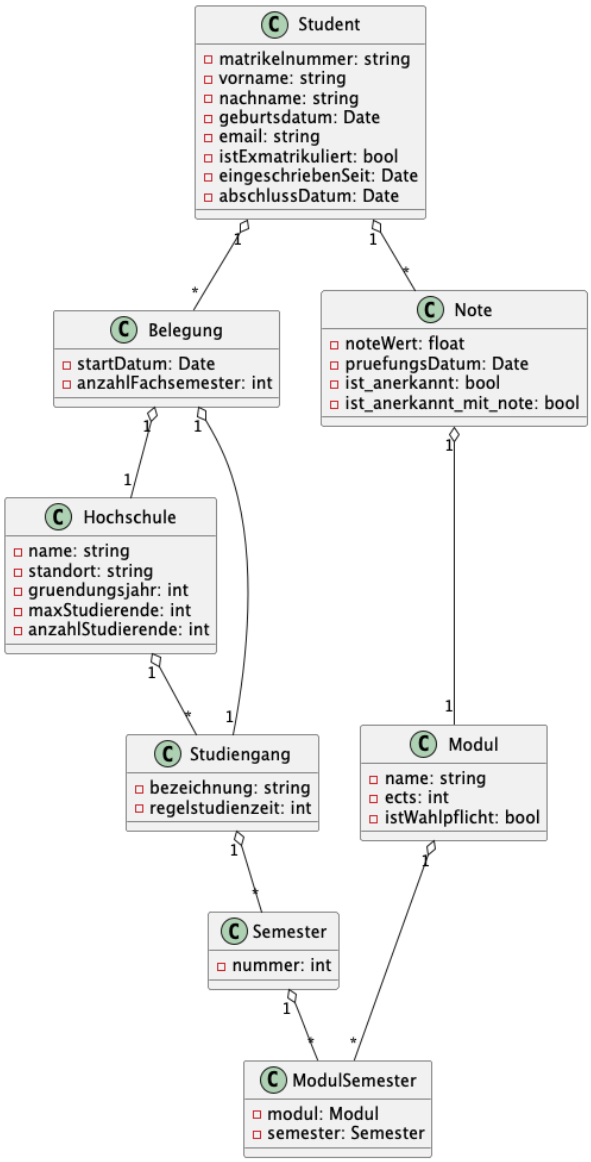
- Im ersten Ortsschild erscheinen – per Python generiert – **Name des Studierenden** und **Studiengang**.
- Ein animiertes **Auto-Icon** bewegt sich semesterweise über den blau eingezeichneten Pfad.
- Ein Klick auf das Auto öffnet ein **Popup**, das den aktuellen Stand der oben definierten Ziele zeigt. Die Textbausteine werden dynamisch aus einem Python-Dictionary geladen, sodass Hinweise und Motivationsmeldungen exakt zum jeweiligen Noten- oder Zeitfortschritt passen.

Damit ist sichergestellt, dass alle relevanten Informationen kompakt, intuitiv und jederzeit abrufbar präsentiert werden.





3. Erstellung eines Klassendiagrammes



Im Rahmen der Modellierung konzentrierte ich mich auf die wesentlichen Entity-Klassen, die den organisatorischen und studienrelevanten Aufbau einer Hochschule abbilden. Ziel war es, ein möglichst klares, strukturiertes und nachvollziehbares UML-Klassendiagramm zu entwickeln, das sich an den Vorgaben der Aufgabenstellung für Phase 1 orientiert. Zentral steht dabei die Klasse Student, die durch Attribute wie Matrikelnummer, Vorname, Nachname und weitere personenbezogene Angaben definiert ist. Sie ist über eine 1:n-Beziehung mit der Klasse Belegung verbunden, die angibt, welchen Studiengang der Studierende belegt und wann die Einschreibung erfolgte.
Ein Studiengang ist in Semester gegliedert, und über die Zwischentabelle ModulSemester werden den einzelnen Semestern Module zugeordnet. Dabei wurde bewusst eine eigene Klasse ModulSemester verwendet, um die flexible Zuordnung von Modulen zu mehreren

Semestern zu ermöglichen.
Zusätzlich existiert die Klasse Note, welche die Zuordnung von Prüfungsleistungen zu Studierenden und Modulen beschreibt. Dabei wurde auch berücksichtigt, dass Leistungen z. B. anerkannt oder mit Note übernommen werden können – daher die entsprechenden booleschen Felder.
Die Klasse Hochschule beschreibt schließlich grundlegende Informationen zur Institution selbst und ist über Aggregationen mit den Studiengängen und Belegungen verbunden. Insgesamt legte ich bei der Modellierung Wert darauf, Redundanzen zu vermeiden, Beziehungen korrekt über Zwischentabellen abzubilden (v. a. bei ModulSemester) und das Diagramm möglichst realitätsnah an der Studienrealität auszurichten. Erweiterungen – z. B. zur Verwaltung von Wahlpflichtmodulen oder detaillierten Studienverläufen – wären im späteren Verlauf denkbar, wurden hier aber bewusst auf das Nötigste reduziert.

4. Machbarkeitsüberprüfung und Erproben von Umsetzungsmöglichkeiten mit Python

Für die Machbarkeits- und Technologieprüfung setzt ich bewusst auf ein schlankes, plattformübergreifendes Python-Ökosystem, das ohne proprietäre Hürden unter Windows, macOS und jeder gängigen Linux-Distribution läuft: Entwickelt wird in den EDU-Versionen PyCharm 2025.1.2 und DataGrip 2025.1.3, der Code liegt auf GitHub und läuft später containerisiert unter Docker 24 auf meinem Unraid-Server. Die Weboberfläche basiert auf Flask 3.1.1 mit Jinja²-Templates sowie Bootstrap 5.3 und der Roboto-Web-Font – so ist sie sofort responsiv, sauber und unabhängig vom Host-Betriebssystem. Als persistente Schicht setzte ich bewusst auf eine dateibasierte SQLite 3.45 statt Excel: echte SQL-Joins, Fremdschlüssel, transaktionale Sicherheit und komplett Docker freundlich. Für die Pfaderkennung kommt OpenCV 4.11 zum Einsatz; Passwort-Hashes und ORM folgen in Phase 2 mit bcrypt 4.3 und SQLAlchemy 2.0. Damit mein Dozent das Ganze schon jetzt öffentlich testen kann, klemme ich einen Cloudflare-Tunnel davor: Die Domain zeigt via Cloudflare-DNS auf das Tunnel-Backend, der cloudflared-Container auf Unraid bindet meine lokale Flask-Instanz (Port 5000) per mTLS ein, und Cloudflare liefert automatisch ein gültiges HTTPS-Zertifikat – ganz ohne offene Ports oder preisgegebene Heim-IP. So bleibt das Dashboard weltweit erreichbar, während ich intern weiter über Tailscale oder localhost entwickle.

Code Beispiele:

Listing 1 - Basis – Flask – Route

Dieses Snippet zeigt, wie die Startseite des Dashboards gerendert wird und verdeutlicht, dass bereits Variablen ins Template (image_webp) injiziert werden.

```
from flask import Flask, render_template
app = Flask(__name__)
@app.route('/')
def dashboard():
	return render_template('index.html',
		image_webp='Infotainment.webp')
if __name__ == '__main__':
app.run(debug=True)
```

Listing 2 – DB – Initialisierung

Dieses Listing legt nur das Schema an; Insert- und Trigger-Logik folgt in Phase 2.

```
import sqlite3, pathlib
DB = pathlib.Path('app.db')
def init_db():
	with sqlite3.connect(DB) as con: con.execute
	('''CREATE TABLE IF NOT EXISTS student(id INTEGER PRIMARY KEY, name
TEXT)''')
	con.execute('''CREATE TABLE IF NOT EXISTS car_pos(student_id INT, semester
REAL, x REAL, y REAL, angle REAL, flip INT)''')
```

Listing 3 – Fetch – API – Endpoint

Liefert Position des Auto-Icons für eine Studenten-/Semester-Kombination

```
@app.route('/api/car-position/<int:stud>/<float:sem>')
def api_car(stud, sem):
	with sqlite3.connect(DB) as con:
		cur = con.execute('SELECT x,y,angle,flip FROM car_pos WHERE
student_id=? AND semester=?', (stud, sem))
		row = cur.fetchone() or (0.6, 0.8, 0, 0)
	return {'success': True, 'x_percent': row[0], 'y_percent': row[1], 'angle':
row[2], 'flip': bool(row[3])}
```