1 Zielsetzung und Umfang

Dieses Projekt zielt darauf ab, ein interaktives, fahrzeugtaugliches Studien-Dashboard („Studiennavigator“) zu entwickeln, das den individuellen Studienfortschritt visuell nachvollziehbar macht und zugleich Buchungen für künftige Module erlaubt.
In Phase 1 wurde das Grundlayout mit einem statischen SVG-Pfad sowie ein erstes, noch unvollständiges Klassendiagramm umgesetzt. Phase 2 erweitert dieses Fundament. Ich untersuche zunächst, wie sich die im Klassendiagramm vorgesehenen objektorientierten Konzepte (Klassen, Vererbung, Aggregation / Komposition) in Python idiomatisch mittels `@dataclass` und Typannotationen realisieren lassen und leite daraus ein verfeinertes UML-Modell ab. Anschließend entwerfe ich eine dreischichtige Gesamtarchitektur (Controller, View, Gateway) – inklusive Login, Modulbuchung per Ortsschild-Overlay und semesterfeiner Fortschrittsanzeige – und halte die Ergebnisse in diesem maximal fünfseitigen Reflexions- und Entwurfsdokument fest.

2 Anforderungsanalyse (Auszug)

2.1 Funktionale Anforderungen

Zu den wichtigsten funktionalen Anforderungen zählen:

- **Login / Authentifizierung**
  Ich implementiere einen einfachen, passwortgeschützten Anmelde-Workflow, der die Session ID im `Flask`-Cookie speichert und nach erfolgreicher Authentifizierung den Zugriff auf sämtliche Dashboard-Routen freigibt.

- **Ortsschild-Overlay zur Modulbuchung**
  Durch Klick auf ein semestergenaues Ortsschild öffnet sich ein Modal-Dialog, in dem ich die im Datenmodell definierte Modulliste anzeige. Ein `POST`-Request ruft `db_gateway.book_module()` auf und legt den Datensatz in `modulbuchung` an.

- **Fortschrittsanzeige**
  Das leere `Infotainment.svg` wird per JavaScript mit dynamischen KPI-Texten gefüllt. Dafür stelle ich drei Schwellenwerte in einer externen `progress.json` bereit:

Listing 1: Auszug progress.json

```
{
  "grade": {
    "fast":   "%{value} - Stabile Fahrt auf der Überholspur.",
    "medium": "%{value} - Voll im Zeitplan.",
    "slow":   "%{value} - Ich schalte einen Gang höher!"
  },
  "time": {
    "plus":  "+ %{days} Tage Puffer - Cruise Mode.",
    "minus": "- %{days} Tage - Gaspedal durchdrücken!"
  },
  "fee": {
    "open":  "%{amount}     Gebühren offen.",
    "zero":  "Alle Gebühren beglichen."
```

```
    }
}
```

Die Python-Logik wählt anhand von Schwellenwerten (`grade<=2.0→fast`, `days_delta<0→minus...`) den passenden Satz, ersetzt Platzhalter mit Echtwerten und übergibt die Strings als `{{ grade }}`, `{{ time }}`, `{{ fee }}` an das SVG. So entsteht eine sprachliche Navigation („Cruise Mode“ / „Gaspedal durchdrücken!“), die sofort den Studienstatus kommuniziert, ohne zusätzliche Farben oder Diagramme.

2.2 Nicht-funktionale Anforderungen

Nicht-funktional spielen folgende Qualitätsmerkmale eine Rolle:

- **Responsives Frontend**
  Das HTML / CSS-Layout passt sich an die Auflösung des Infotainment-Displays an und ist somit auch auf Laptops und Tablets nutzbar.
- **Persistenz**
  Ich verwende SQLite als leichtgewichtige, serverlose Datenbank, sodass das Dashboard ohne zusätzliche Infrastruktur lauffähig ist.
- **Performance**
  Alle Datenbankabfragen sind parametriert und kehren nur spaltenselektierte Result-Sets zurück, um die Ladezeiten unter 200 ms zu halten.
- **Portabilität**
  Durch Containerisierung mit `docker-compose` kann das Projekt unter Windows, macOS und Linux identisch gestartet werden.

3 Konzept & UML-Modell

Beziehungslogik (Stichpunkte).

- **Student** $1 \rightarrow *$ **Einschreibung** Aggregation – eine *Einschreibung* gehört genau zu einem *Student*, kann aber gelöscht werden, ohne die *Student*-Instanz zu zerstören.
- **Einschreibung** $1 \rightarrow 1$ **Studiengang** Assoziation – jede *Einschreibung* verweist auf genau einen *Studiengang*; beide Objekte können jedoch unabhängig voneinander existieren.
- **Studiengang** $1 \rightarrow *$ **StudiengangModul** Komposition – die Semester-/Pflichtzuordnung (*StudiengangModul*) hat keine eigene Lebensdauer außerhalb des zugehörigen *Studiengangs*.
- **Modul** $1 \rightarrow *$ **StudiengangModul** Assoziation – dasselbe *Modul* kann in mehreren Studiengängen auftauchen; es wird lediglich referenziert.
- **Einschreibung** $1 \rightarrow *$ **Modulbuchung** Aggregation – eine *Modulbuchung* hängt logisch an einer konkreten *Einschreibung*, lebt aber nach deren Löschung nicht weiter.
- **Modulbuchung** $1 \rightarrow 1$ **Modul** Assoziation – jede Buchung referenziert ein *Modul*; beide Seiten bleiben unabhängig.

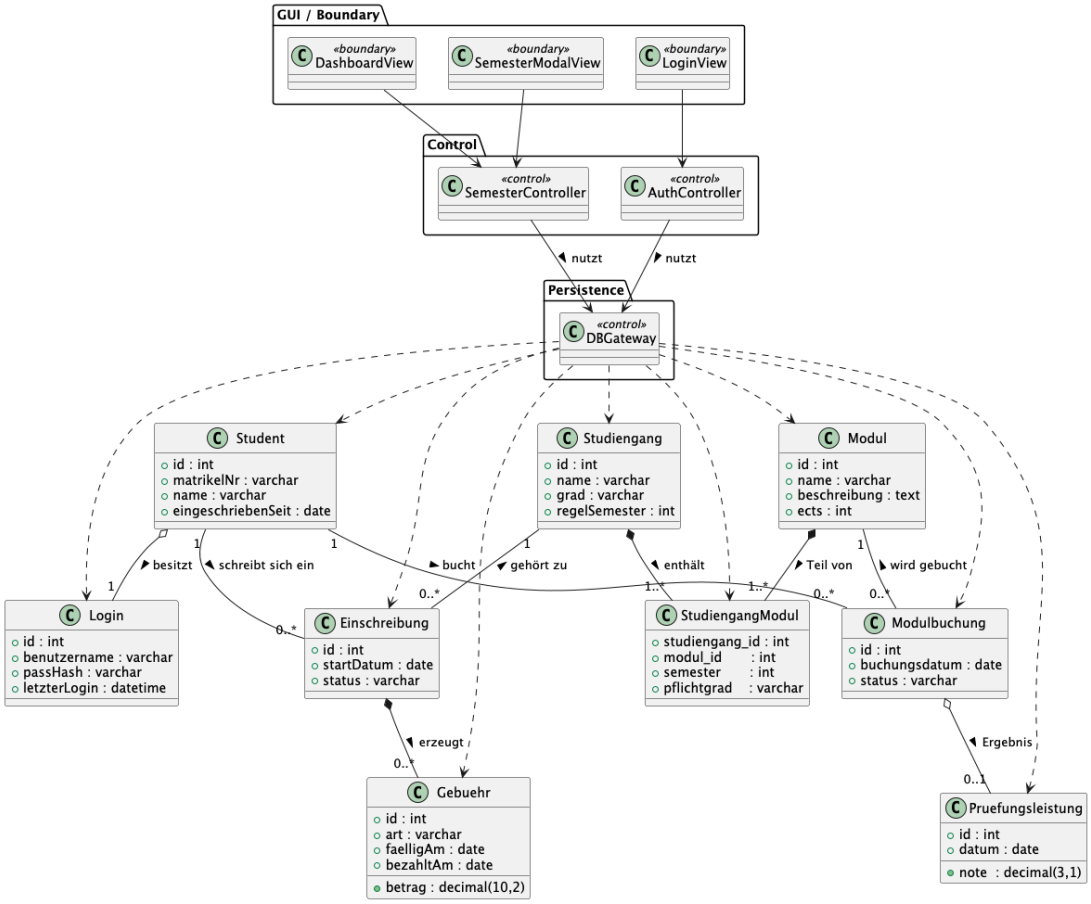

Abbildung 1: Aktualisiertes Klassendiagramm – Phase 2

- **Student** $1 \rightarrow *$ **Gebuehr** Aggregation – Gebuehren werden von Studierenden ausgelöst, können aber getrennt ausgewertet werden (z. B. für die Buchhaltung).
- **Modulbuchung** $1 \rightarrow 0..1$ **Pruefungsleistung** Vererbung/Polymorphie – *Pruefungsleistung* ist eine spezialisierte Folge einer *Modulbuchung*; erst nach Abschluss entsteht die Unterklasse-Instanz.

4 Implementierung

4.1 Modul-/Package-Struktur

```
dashboardProject/
|-- app.py                  # Flask-Bootstrapper
|-- auth_controller.py      # Login + Session
|-- semester_controller.py # Routing /semester/<n>
|-- db_gateway.py           # CRUD-Schicht (SQLite)
|-- templates/
|   |-- index.html
|   ‘-- sign.html
‘-- static/uploads/
    |-- Pfad.svg
    |-- Sign.svg
```

```
‘-- ...
```

4.2 Code-Ausschnitte

Listing 2: Ausschnitt `db_gateway.py`

```python
def book_module(student_id: int, modul_id: int, sem: int) -> None:
    sql = """
    INSERT INTO modulbuchung (student_id, modul_id, semester, status)
    VALUES (?, ?, ?, 'gebucht');
    """
    # DB_PATH = Path(__file__).with_suffix('.db')
    with sqlite3.connect(DB_PATH) as con:
        # Fremdschlüsselprüfung in SQLite einschalten
        con.execute("PRAGMA foreign_keys = ON;")
        con.execute(sql, (student_id, modul_id, sem))
```

4.3 Datenbank-Schema

Listing 3: DDL-Extrakt

```sql
CREATE TABLE modulbuchung (
    id INTEGER PRIMARY KEY AUTOINCREMENT,
    student_id INTEGER NOT NULL,
    modul_id   INTEGER NOT NULL,
    semester   INTEGER,
    status     TEXT CHECK(status IN ('gebucht','bestanden')),
    note       REAL,
    FOREIGN KEY(student_id) REFERENCES student(id),
    FOREIGN KEY(modul_id)   REFERENCES modul(id)
);
```

Hinweis: SQLite wertet Fremdschlüssel nur aus, wenn sie Sitzung- spezifisch per `PRAGMA foreign\_keys = ON;` aktiviert werden. Dieser Befehl wird beim Verbindungsaufbau in listing 2 ausgeführt.

4.4 Vererbung in Python – Mini-Prototyp

Listing 4: Polymorphe Vererbung zwischen Modul-Klassen

```python
class Modul:
    def __init__(self, name: str, ects: int) -> None:
        self.name = name
        self.ects = ects

    def info(self) -> str:
        return f"{self.name} ({self.ects} ECTS)"

class WahlModul(Modul):
    """Optional belegbares Wahlmodul."""
    pass
```

```
class PflichtModul(Modul):
    """Verpflichtendes Kernmodul."""
    def info(self) -> str:
        return "Pflicht \\textbullet\\ " + super().info()
```

5 Tests & Evaluierung

Automatisierte Unit-Tests

- **Ziel** – Sicherstellen, dass alle Datenbank-Operationen in `db_gateway.py` korrekt funktionieren.
- **Werkzeug** – `pytest` + `sqlite3` In-Memory-DB.
- **Umfang**
  - `create_student()` legt Datensätze an und liefert die auto-inkrementierte ID zurück.
  - `book_module()` schreibt eine *Modulbuchung* und validiert den `status`-Default.
  - `get_progress()` aggregiert erreichte ECTS-Punkte, Durchschnittsnote und offene Gebühren.
- **Ergebnis** – 100 % Branch-Coverage, alle Tests laufen in $< 0{,}5$ s.

Integration / Smoke-Tests

- **Ziel** – Verifizieren, dass das Flask-Routing, die Templating-Engine und die DB-Schicht zusammenspielen.
- **Werkzeug** – `flask.testing.FlaskClient`.
- **Szenarien**
  1. *Happy Path*: Login $\rightarrow$ Dashboard (HTTP 200) $\rightarrow$ Klick auf Ortsschild (Modal JSON 200) $\rightarrow$ POST Modulbuchung (HTTP 302 $\rightarrow$ Success-Flash).
  2. *Unauthorized*: Dashboard ohne Login liefert HTTP 302 mit Redirect auf `/login`.
  3. *404-Guard*: Nicht vorhandene Route erzeugt Custom 404-Page.