Objektorientierte und funktionale Programmierung mit Python (DLBDSOOFPP01_D)
Aufgabenstellung: Studien-Dashboard (OOP Python)
Name: Gökhan Sen / Matrikelnummer: IU14140216

Phase 02 – (Erarbeitungs-/Reflexíonsphase)

1. Ausgangslage und Zielsetzung der Phase 2

In Phase 1 wurde ein fachliches UML-Entity-Modell (Studiengang, Semester, Modul, ModulBelegung, Student) sowie eine erste Machbarkeitsprüfung (SQLite, einfache GUI, UML-Tooling) erstellt. Ziel der Phase 2 ist die Reflexion und der Entwurf der prototypischen Umsetzung: • Untersuchung, wie die relevanten objektorientierten Konzepte in Python umgesetzt werden können. • Ableitung einer Gesamtarchitektur für den späteren Dashboard-Prototypen inkl. vollständigem UML-Klassendiagramm und textueller Erläuterung.

1.1 Präzisierung der Dashboard-Ziele (Tutor-Feedback)

Ziel der Anwendung ist die Darstellung des Studienfortschritts und der Durchschnittsnote als zentrale KPIs des Dashboards.

Im Tutor-Feedback zu Phase 1 wurde angemerkt, dass der Begriff „Zentrale KPIs zur Studiendauer“ zu allgemein formuliert ist. In Phase 2 werden die KPIs daher konkretisiert und eindeutig hergeleitet.

Die KPIs werden aus Modul-bezogenen Eingangsdaten abgeleitet, insbesondere aus der Ist-Note und dem Bestehensdatum einzelner Module. Der Studienfortschritt wird als kumulative ECTS über die Zeit dargestellt, die Durchschnittsnote als ECTS-gewichtete Durchschnittsnote über die Zeit.

Damit ist transparent nachvollziehbar, wie sich die übergeordneten KPIs aus den einzelnen Modulbelegungen zusammensetzen.

2. Untersuchung objektorientierter Konzepte in Python

Für die prototypische Umsetzung wurden die für das Klassendiagramm relevanten OOP-Konzepte in kleinen Testprogrammen überprüft. Der Fokus liegt dabei nicht auf „viel Code“, sondern auf den Python-spezifischen Besonderheiten (z. B. dynamische Typisierung, dataclasses, duck-typing) und darauf, ob das UML-Modell aus Phase 1 ohne Änderungen abbildbar ist.

2.1 Klassen, Attribute und Initialisierung

In Python können Entity-Klassen klassisch über __init__ oder komfortabel über dataclasses umgesetzt werden. Dataclasses reduzieren Boilerplate (Konstruktor, Repräsentation) und eignen sich

gut für Datenobjekte wie Modul, Semester oder Student. Für Validierung kann __post_init__ genutzt werden.

Erkenntnis: Für reine Entities werden dataclasses verwendet; Validierungen werden in Service/Validation ausgelagert.

2.2 Komposition vs. Aggregation

UML-Komposition (z. B. Studiengang „besteht aus“ Semestern/Modulen) wird in Python typischerweise durch Containment (Listen/Collections) umgesetzt. Lebenszyklus-Kopplung wird eher durch die Anwendung/Repository gesteuert als „sprachlich erzwungen“.

Erkenntnis: Beziehungen werden über IDs/Listen modelliert; Persistenz in SQLite definiert die tatsächliche Kopplung.

2.3 Vererbung, Abstraktion und Interfaces

Für Vererbungsbeziehungen kann Python klassische Inheritance nutzen. Falls ein Interface-Verhalten benötigt wird, eignen sich Abstract Base Classes (abc.ABC) oder Protocols (typing.Protocol). Für dieses Projekt sind Entities überwiegend flach; wichtige Erweiterungspunkte liegen eher bei Repositories/Services als bei Entity-Vererbung.

Erkenntnis: Vererbung ist optional; stattdessen klare Schichten (Repository/Service/UI).

2.4 Properties, berechnete Kennzahlen und Methoden

Berechnete KPIs (z. B. Ist-Durchschnittsnote) werden als Methoden im Service oder (weniger empfehlenswert) direkt im Studiengang modelliert. In Python können Properties (@property) eine saubere, attributähnliche API bieten, jedoch sollte die Datenbasis (DB) nicht unkontrolliert in Entities „hineinleaken“.

Erkenntnis: KPI-Berechnung im Service-Layer; Entities bleiben schlank und testbar.

2.5 Persistenz & Repository-Pattern

SQLite wird über sqlite3 angebunden. Um UI/Business-Logik von SQL zu trennen, bietet sich ein Repository-Layer an (CRUD-Methoden pro Entity). Das erleichtert Tests und reduziert Kopplung.

Erkenntnis: Repository-Layer wird Teil der Gesamtarchitektur (zusätzliche Klassen gegenüber Phase 1).

2.6 Ergebnis der Untersuchung

Das Entity-UML aus Phase 1 ist in Python gut umsetzbar. Für die prototypische Umsetzung sind jedoch zusätzliche Klassen sinnvoll (Repository/Service/UI/Validation). Diese gehören nicht in das Phase-1-Entity-Diagramm, sind aber Bestandteil der Phase-2-Gesamtarchitektur.

3. Ableitung der Gesamtarchitektur für den Prototypen

Basierend auf den Erkenntnissen aus Abschnitt 2 wird eine Schichtenarchitektur gewählt: • UI-Schicht: Benutzerinteraktion (GUI oder CLI) • Service-Schicht: Fachlogik/KPI-Berechnung, Validierung, Use-Cases • Repository-Schicht: Persistenzzugriff (SQLite) • Model-Schicht: Entities/DTOs

3.1 Zusätzliche Klassen gegenüber Phase 1

In der Phase-2-Architektur werden neben den Entity-Klassen zusätzliche Klassen eingeführt:

- Database / ConnectionFactory (zentraler Zugriff auf SQLite-Verbindung und Schema-Init).
- Repositories (z. B. ModulRepository, ModulBelegungRepository).
- Services (z. B. StudiengangService/KpiService für Fortschritt, Durchschnittsnote, Studiendauer).
- Validation (z. B. Eingabeprüfung Note, Datum, ECTS).
- UI (z. B. Tkinter-Form für CRUD auf ModulBelegung).

4. Gesamtarchitektur als UML-Klassendiagramm (Prototyp)

Die Architektur folgt einem schichtenorientierten Ansatz. Die UI-Schicht übernimmt ausschließlich die Benutzerinteraktion und ruft definierte Use-Cases der Service-Schicht auf. Die Service-Schicht kapselt die fachliche Logik, die Berechnung zentraler Kennzahlen sowie die Validierung von Eingaben. Die Repository-Schicht abstrahiert den Zugriff auf die SQLite-Datenbank und stellt CRUD-Operationen für die Domänenobjekte bereit. Die Model-Schicht enthält die fachlichen Entitäten des Systems. Durch diese Trennung der Verantwortlichkeiten (Separation of Concerns) bleibt die Anwendung wartbar, testbar und gut erweiterbar, hinsichtlich Phase 3. Das in Phase 1 entwickelte fachliche UML-Klassendiagramm bleibt unverändert gültig. Das folgende Diagramm erweitert dieses Modell um technische Klassen (Service-, Repository- und UI-Schicht), um die Gesamtarchitektur des Prototyps darzustellen.

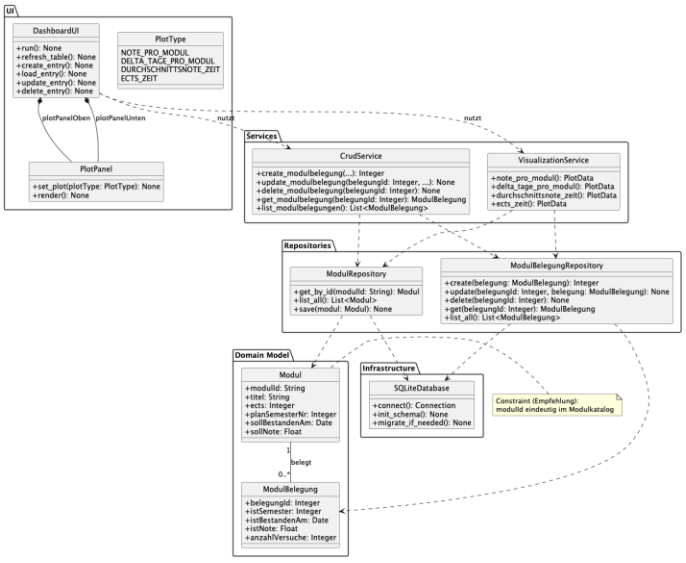

Abbildung 1: UML-Klassendiagramm der Gesamtarchitektur (Phase 2)

4.1 Textuelle Erläuterung der Architektur

Die UI ruft Use-Cases in der Service-Schicht auf (z. B. „Belegung speichern“, „Belegung ändern“, „Belegung löschen“). Der Service validiert Eingaben und nutzt Repositories für den Datenzugriff. Repositories kapseln SQL und geben Entities/DTOs zurück. KPI-Berechnungen greifen auf Modul- und Belegungsdaten zu und liefern Kennzahlen zurück, die im Dashboard angezeigt werden.

Die UML-Darstellung der Gesamtarchitektur des Systems wurde bereits in Kapitel 4 vorgestellt (siehe Abbildung 1). Das dort dargestellte Diagramm integriert sowohl das fachliche Domänenmodell aus Phase 1 als auch die technische Architektur der Phase 2 und bildet damit die Grundlage für die prototypische Umsetzung.

4.2 Einordnung der Diagramme in die Architektur

Die Diagramme sind integraler Bestandteil der Benutzeroberfläche und werden direkt in die GUI eingebettet. Hierzu werden zwei PlotPanels verwendet, die jeweils unterschiedliche Diagrammtypen darstellen können. Die Auswahl der darzustellenden Diagramme erfolgt über das PlotType-Enum, welches folgende Visualisierungen umfasst: Note pro Modul, Zeitabweichung pro Modul, Durchschnittsnote über Zeit, ECTS-Fortschritt über Zeit

Diese Struktur ermöglicht eine klare Trennung zwischen Benutzerinteraktion, fachlicher Berechnung und Darstellung, wie im UML-Architekturdiagramm dargestellt.4.3 Abgrenzung fachliches Modell vs. technische Architektur

Die Model-Klassen repräsentieren die fachlichen Kernobjekte des Systems. Jede Entität besitzt eine eindeutige Identität und kapselt ihre fachlichen Attribute. Die Repository-Klassen übernehmen die vollständige CRUD-Verantwortung für die Persistenz. Sie kapseln SQL-Zugriffe und geben fachliche Objekte an die Service-Schicht zurück. Die Service-Schicht implementiert die Anwendungslogik, z. B. die Berechnung von Studienfortschritt, Durchschnittsnote und Studiendauer. Zusätzlich werden hier fachliche Validierungen durchgeführt. Die UI-Schicht dient ausschließlich der Interaktion mit der Benutzerin bzw. dem Benutzer. Sie enthält keine Fachlogik, sondern delegiert alle Operationen an Services.

4.4 Textuelle Erläuterung der UML-Architektur

Abbildung 1 zeigt das UML-Klassendiagramm der Gesamtarchitektur für den Prototypen. Es erweitert das fachliche Entity-Diagramm aus Phase 1 um technische Klassen für Persistenz, Service-Logik und Benutzerinteraktion.

4.5 UML-Klassendiagramm – Gesamtarchitektur (Prototyp)

Die Anwendung ist in vier logisch getrennte Schichten gegliedert: UI (Benutzerinteraktion), Services (Fachlogik, KPI-Berechnung, Validierung), Repositories (Datenzugriff und Persistenz via SQLite) und Model (fachliche Entitäten wie Student, Studiengang, Modul, ModulBelegung).
Die UI greift ausschließlich auf Services zu; Services kapseln fachliche Regeln und nutzen Repositories für den Datenzugriff. Die Model-Schicht ist frei von Persistenz- und UI-Logik.

4.6 Architekturüberblick (Schichtenmodell)

Die Gesamtarchitektur des Systems dient dazu, das in Phase 1 entwickelte fachliche Modell in eine wartbare, erweiterbare und testbare Softwarestruktur zu überführen. Hierzu wird eine klar geschichtete Architektur gewählt, die Verantwortlichkeiten trennt und Abhängigkeiten minimiert.

5. Reflexion und Ausblick

Die Untersuchung zeigt, dass das fachliche Modell aus Phase 1 in Python umsetzbar ist, aber für einen stabilen Prototypen zusätzliche Klassen für Persistenz, Service-Logik und UI erforderlich sind. Die Phase-2-Architektur reduziert Kopplung und erleichtert Tests sowie spätere Erweiterungen. Ausblick: In der Finalisierungsphase (Phase 3) werden UI/Usability weiter verbessert, die Visualisierung ausgebaut und der Code in einem vollständigen Projektverzeichnis inkl. Installationsanleitung bereitgestellt.

Die dargestellte Architektur stellt den konzeptionellen Stand dar und kann im Rahmen der Implementierung weiter konsolidiert oder vereinfacht werden.