

### **Phase 3 - Abstract – Studien-Dashboard**

#### **Ausgangslage und Zielsetzung**

Im Portfoliokurs wurde ein Studien-Dashboard als Python-Prototyp umgesetzt. Ziel ist ein schneller Überblick über den Studienstand anhand zweier KPIs: (1) Studienfortschritt und (2) Durchschnittsnote. Auf Basis des Tutor-Feedbacks wurden die Ziele in Phase 2 präzisiert: Fortschritt als kumulative ECTS über die Zeit und Durchschnittsnote als ECTS-gewichtete Note über die Zeit. Dadurch bleiben Kennzahlen aus Modulbelegungen jederzeit nachvollziehbar.

#### **Vorgehensweise und Evolution Phase 1 bis Phase 3**

Phase 1 fokussierte ein sprachunabhängiges UML-Entity-Modell (Student, Studiengang, Modul, ModulBelegung) und eine konzeptionelle KPI-Sicht. Zudem wurden Persistenz, GUI-Grundlagen und UML-Tooling auf Machbarkeit geprüft.

In Phase 2 wurde das Modell auf Python abgebildet (dataclasses, Validierung, Repository-Pattern) und eine Schichtenarchitektur (UI/Service/Repository/Model) abgeleitet, um Verantwortlichkeiten zu trennen und die Lösung wart- und testbar zu gestalten.

Phase 3 überführte die Architektur in ausführbaren Code (UML: Phase03/docs/uml). Das Domänenmodell wurde pragmatisch gestrafft (kein persistiertes Semester-Entity, Semesternummern als Attribute). KPI-Logik wurde aus Entities in die Service-Schicht verlagert, um Entities schlank zu halten und Geschäftslogik zentral zu bündeln.

#### **Umsetzung im Prototyp (Funktionsumfang)**

Der Prototyp bietet eine Tkinter-GUI zur Dateneingabe und KPI-Visualisierung. Im Fokus steht CRUD für Modulbelegungen (Bestanden-Datum, Note, Versuche) sowie die Pflege von Studiengang-Stammdaten (z. B. Startdatum, Soll-Semester, Ziel-Note). Die Daten werden in SQLite persistiert. Diagramme zeigen u. a. ECTS-Fortschritt, Durchschnittsnote und Noten pro Modul sowie Plan-/Ist-Abweichungen.

## Architektur und technische Entscheidungen

Die Anwendung ist in vier Schichten gegliedert:

- UI: Eingabe, Tabellenansichten, Diagramme (keine Fachlogik).
- Service: Use-Cases, Validierung, KPI-Berechnung, Datenreihen für Plots.
- Repository: Kapselung aller SQL-Zugriffe (CRUD und KPI-Abfragen).
- Model: fachliche Entities/DTOs.

Die Abhängigkeitsrichtung ist „nach unten“ umgesetzt: UI kennt nur Services, Services nutzen Repositories, Repositories greifen auf die Datenbank zu. Der DB-Zugriff ist zusätzlich über ein kleines Protocol/Interface abstrahiert, um Infrastruktur zu entkoppeln und Tests zu erleichtern.

## Reflexion, Erkenntnisse und Weiterverwendung

Gut funktionierte die iterative Herleitung von einer fachlichen Sicht (UML) über eine technische Architektur bis hin zur Implementierung. Klare Trennung der Verantwortlichkeiten erleichtert die Weiterentwicklung: GUI-Änderungen betrafen nicht die SQL-Logik, und KPI-Anpassungen konnten zentral im Service erfolgen. Die Entscheidung, Kennzahlen aus gespeicherten Modulbelegungen abzuleiten (statt sie manuell zu pflegen), erhöhte Konsistenz und Nachvollziehbarkeit.

Aufwändig war die GUI-Integration im Detail (robuste Eingabevalidierung, verständliche Fehlermeldungen, konsistentes Aktualisieren der Anzeige nach CRUD-Aktionen). Außerdem erfordert das Zusammenspiel aus Persistenz, Service-Logik und Visualisierung eine disziplinierte Schnittstellengestaltung, damit die Schichten nicht vermischen.

Wesentliche Erkenntnis ist, dass ein fachliches UML-Entity-Modell in Python gut abbildbar ist, ein wartbarer Prototyp jedoch zusätzliche technische Bausteine benötigt (Services, Repositories, Validierung, DB-Abstraktion). Besonders stolz bin ich auf die konsequente Abhängigkeitsrichtung (UI → Service → Repository → DB) und die umfassende Quelltextdokumentation, die dem Tutor ein schnelles Verständnis und Testen ermöglicht.

Weiterentwicklung: automatisierte Tests (Service/Repository), Import/Export (z. B. CSV/myCampus) sowie zusätzliche Diagramme und Prognosefunktionen.