# Reflexions- und Entwurfsdokument

## „Entwicklung eines Dashboards für dein Studium“

Kursbeschreibung:

DLBDSOOFPP01_D – „Objektorientierte und funktionale Programmierung mit Python“

Studiengang:

Angewandte Künstliche Intelligenz, Bachelor of Science (B.Sc.)

Verfasser:

Phillip Riemer

Solmsstraße 25

60486 Frankfurt am Main

phillip.riemer@iu-study.org

Matrikelnummer:

IU14128175

## 1. Untersuchung der Umsetzung objektorientierter Konzepte in Python

Das Ziel der Reflexions-Phase ist es, das in der Konzeptionsphase erarbeitete objektorientierte Modell zur Studienüberwachung prototypisch in Python zu testen und zu implementieren. Das ursprüngliche UML-Diagramm hat versucht die komplexen Beziehungen zwischen den Entity-Klassen aus der Realität zu berücksichtigen. Es hat zahlreiche Beziehungen enthalten, unter anderem zwischen Student, Studiengang, Semester, Modul, Prüfungsleistung und sogar einer steuernden Zeitplan-Klasse. Während diese Struktur modelltheoretisch sinnvoll erschien, hat sich in ersten Programmier-Experimenten gezeigt, dass sie in der Praxis im Hinblick auf Wartbarkeit, Erweiterbarkeit und Benutzerinteraktion eher hinderlich war. Viele Beziehungen führten zu starker Kopplung zwischen Klassen, was die Instanziierung unnötig erschwerte, ohne Mehrwert beizutragen.

Deshalb bin ich anschließend bewusst schrittweise vorgegangen und habe alle Konzepte in kleinen, modularen Testprogrammen ausprobiert, bevor ich sie in das Gesamtsystem integriert und darauf aufgebaut habe. Diese Vorgehensweise hat mir sehr geholfen, Fehler frühzeitig zu erkennen und schnell zu beheben. Ich konnte zum Beispiel durch die Aufteilung in unabhängige Module gezielt testen, ob eine Komposition korrekt funktioniert, ob die Datumsverarbeitung fehlerfrei läuft oder ob die Methoden zur Datenkapselung wie gewünscht arbeiten.

Auf das Prinzip der Modularität habe ich besonderen Wert gelegt. Jede Klasse erfüllt eine klar abgegrenzte Aufgabe und lässt sich isoliert testen. Eine Klasse „Zeitplan“ wird durch die Verwendung der datetime-Funktion in Python nicht mehr benötigt. Diese erlaubt die einfache Berechnung von Zeiträumen, ohne dass benötigt wird. Auch die Klasse Semester konnte entfernt werden. Da der Student (ich) alle Module hintereinander absolviert und diese keinem Semester fest zugeordnet sind, hatte eine eigene Klasse Semester keinen Mehrwert. Die Zuordnung eines Moduls zum jeweiligen Semester erfolgt später erst dann nach Zeiträumen, wenn es in Diagrammen benötigt wird. Dadurch konnte ich auf Strukturen mit überflüssiger Komplexität verzichten und trotzdem die volle Funktionalität sicherstellen.

In der überarbeiteten Version des Klassendiagramms konnte ich die Anzahl der Klassen und die Beziehungen auf das Wesentliche reduzieren, was für sinnvolle Ein- und Ausgaben im Dashboard tatsächlich benötigt wird: die Klassen Student, Studiengang, Modul und Prüfungsleistung. Auf Beziehungsklassen, wie Assoziationsklassen, habe ich zugunsten der Komplexitätsreduktion bewusst verzichtet. Student steht in der Realität, so auch im UML-Diagramm, in Assoziationsbeziehungen mit Modul und Prüfungsleistung. In der Funktionalität dieses Prototyps nutzt die Klasse diese Informationen jedoch nicht. Die Verknüpfung im Code hätte keinen Mehrwert erzeugt, weil das Dashboard nur Daten von genau einem Studenten anzeigt.  Die Klasse Studiengang ist dem Studenten durch Komposition zugeordnet. Das heißt, bei der Erstellung der Klasse Studiengang wird die Klasse Student benötigt. Genauso verhält es sich mit der Klasse Prüfungsleistung, die ein Modul benötigt. Diese Form der Objektverknüpfung ist ein zentrales Prinzip meiner objektorientierten Modellierung und verdeutlicht die starke Abhängigkeit der Klassen auf den Lebenszyklus der anderen.

Im Code wurden die Kompositionen über eine Eingabe des Konstruktors und Referenz im Attribut wie folgt gelöst. Ein Beispiel gibt die Klasse Studiengang, welche die Klasse Student verwendet:

```
class Studiengang:
    def __init__(self, eingabe_student: Student, …):
        self.student = eingabe_student
        …
```

Ein weiteres Prinzip, um Klassen und Eingaben so einfach wie möglich zu halten, ist die Kapselung. Alle verwendeten Klassen besitzen klar definierte Konstruktoren, in denen die internen Attribute initialisiert werden. Nach außen hin werden diese Daten ausschließlich über die Property-Methoden „daten()“ zugänglich gemacht. Damit nutzen die Klassen teilweise einen Factory- und Abstraktions-Ansatz, so muss die interne Verarbeitung bei späterer Verwendung nicht jedes Mal offengelegt sein, kann aber bei Bedarf jederzeit in der Klasse geändert werden, ohne dass dies Auswirkungen auf andere Teile des Programms hat. So rechnet etwa die Klasse Prüfungsleistung im Hintergrund automatisch aus, wie viele Tage Vorbereitung der Prüfling ab Modulstart hatte. Der Benutzer muss diese Berechnung nicht selbst durchführen – die Logik ist gekapselt und damit abstrahiert. Das erhöht die Wiederverwendbarkeit und reduziert die Fehleranfälligkeit deutlich.

Ein Beispiel des Factory-Ansatzes zeigt ebenfalls die Verarbeitung der Attribute in der Klasse Studiengang. Die Eingabe wird automatisch vom Stringformat in ein Datumsformat umgewandelt und ein weiteres Datum daraus berechnet.

```
class Studiengang:
    def __init__(self, …, eingabe_beginn, …):
        …
        self.start_datum = datetime.strptime(eingabe_beginn, "%d.%m.%Y")
        self.ende_datum = self.start_datum + timedelta(days=3*365)
```

Die Datenausgabe der Entity-Klassen erfolgt als Dictionary, durch diese Form der Ausgabe lassen sich später direkt CSV-Dateien befüllen.

Die Entscheidung gegen die Verwendung von Vererbung wurde bewusst getroffen. In einem so datenorientierten Projekt wie diesem hätte eine Vererbungshierarchie keinen echten Mehrwert gebracht, sondern eher zusätzliche Komplexität erzeugt. Stattdessen habe ich mich für Komposition entschieden, da dieses Konzept in Python schnell, transparent und wartbar umgesetzt werden kann.

Die objektorientierte Umsetzung des Dashboards in Python war, auch wenn das ursprüngliche UML-Diagramm nicht vollständig übernommen wurde, erfolgreich. Entscheidend war die methodisch begründete Reduktion auf das Wesentliche. Die eingesetzten Konzepte wie Modularität , Aggregation, Kapselung und Abstraktion wurden konsequent in den Entity-Klassen angewendet und durch Testprogramme verlässlich geprüft.

## 2. Erstellen der Gesamtarchitektur

Meine Gesamtarchitektur des Codes ist modular aufgebaut und umfasst insgesamt vier Schichten: Entity-Klassen zur Datenstrukturierung, einen CSV-Controller zur Datenspeicherung, eine Logikschicht zur Berechnung und Visualisierung sowie eine GUI-Schicht zur Benutzerinteraktion.

Die unterste Ebene bilden die Entity-Klassen: Student, Studiengang, Modul und Prüfungsleistung. Diese vier Klassen beschreiben die Kerndatenstrukturen des Dashboards. Die Klasse Student verwaltet Name und Matrikelnummer, während Studiengang diese um Informationen, wie Studiengangsbezeichnung, Abschlussart sowie Start- und Enddatum des Studiums erweitert. Beide Klassen stehen in einer Kompositionsbeziehung. Ein Studiengang benötigt einen Studenten und verweist in der Konstruktor-Eingabe und mit einem Attribut auf ihn, der Student hingegen kann auch unabhängig bestehen. Entsprechende Logik gilt auch für die Klassen Modul und Prüfungsleistung. Eine Prüfungsleistung referenziert als Konstruktor-Eingabe und als Attribut das Modul, dieses kann jedoch auch für sich alleinstehen. Alle Entity-Klassen stellen ihre Informationen über Property-Methoden zur Verfügung, was die Weiterverarbeitung in CSV-Dateien vereinfacht.

Darauf aufbauend fungiert die Klasse CSV_Controller als zentrale Schnittstelle zur Datenspeicherung. Sie ist verantwortlich für das Schreiben und Überschreiben der beiden CSV-Dateien Student.csv und Module_abgeschlossen.csv. Mit den Methoden setze_student_csv(), füge_modul_csv_hinzu() und lösche_modul_csv() können sowohl neue Daten angelegt als auch bestehende Einträge entfernt werden. Die Einbindung der Entity-Klassen in die CSV-Erzeugung sichert dabei Datenkonsistenz. Eine automatische chronologische Sortierung der CSV-Einträge nach Prüfungsdatum sorgt für korrekte Darstellungen in den späteren Diagrammen und einer Tabelle.

Die darauffolgende Logik- und Auswertungsschicht wird durch die Klasse Plots_Berechnungen realisiert. Sie liest die Daten aus den CSV-Dateien ein und berechnet verschiedene statistische Werte. Dazu zählen unter anderem der Notendurchschnitt und die Abweichung vom Zeitplan, der ein Modul pro Monat für mein Bachelor-Studium vorsieht. Diese Kennzahlen fließen direkt in das Dashboard ein. Die Klasse erstellt ebenfalls mehrere Visualisierungen mittels matplotlib. Zu den Diagrammen zählen ein horizontales Fortschrittsdiagramm für Zeit und ECTS, ein Noten-Histogramm, ein chronologischer Notenverlauf mit gleitendem Durchschnitt, sowie ein Diagramm zur durchschnittlichen Bearbeitungsdauer für ein Modul je Semester. Zusätzlich wird eine kompakte Tabelle mit Moduldaten erzeugt. Damit sind alle ursprünglich geplanten Aspekte des Studienverlaufs darstellbar.

Die letzte Ebene bildet die Benutzeroberfläche in der Klasse GUI_Controller. Diese organisiert das Layout mittels eines Rastersystems mit Containern, die gezielt Inhalte aufnehmen. Textausgaben und Diagramme werden in diesen über Methoden eingebunden. Zusätzlich erlaubt die Oberfläche über Buttons das Hinzufügen und Löschen von Modulen direkt durch den Nutzer. Die eingegebenen Daten werden sofort in die CSV-Dateien übernommen. Das GUI-Fenster schließt und öffnet sich neu

bei Bestätigung der Eingaben durch Button-Klick, so ist sichergestellt, dass Änderungen sofort aktualisiert sichtbar werden. Die GUI-Klasse stellt somit die direkte Interaktionsschnittstelle dar.

Das untenstehende UML-Diagramm veranschaulicht die gesamte Architektur. Die Entity-Klassen stehen im Zentrum und sind wie oben beschrieben mit Aggregationen miteinander verbunden. Die CSV-Controller-Klasse steht in Dependency-Beziehung zu den Entity-Klassen, da sie deren Daten-Methoden zum Beschreiben der CSVs nutzt. Die Logik-Klasse liest diese Daten, verarbeitet sie und erzeugt daraus Auswertungen, die anschließend in der GUI-Klasse eingebunden werden. Auch diese steht in Dependency-Beziehung zur Logik- und CSV-Controller-Klasse. Die Struktur folgt dem Prinzip der Trennung von Datenmodell, Logik und Darstellung. Diese zusätzlichen Klassen wurden bewusst eingeplant, um Modularität und Erweiterbarkeit zu ermöglichen. Der Aufbau gewährleistet, dass einzelne Komponenten austausch- oder testbar sind, ohne das Gesamtsystem zu stören.

Die Architektur zielt auf saubere objektorientierte Modellierung mit einem funktionierenden Prototypen ab, der reale Daten verarbeitet und nutzerfreundlich darstellt. Erweiterungen wären problemlos denkbar, beispielsweise Importfunktionen oder alternative Datenquellen oder Benachrichtigungen.