# Objektorientierte und funktionale Programmierung mit Python (DLBDSOOFPP01_D)

# Reflexionsphase

Studiengang: Softwareentwicklung

Datum: 07. Juli 2025

Name: Yannic Dykow

Matrikelnummer: 3220807

# Reflexions- und Entwurfsdokument

## Dashboard für Dein Studium - Erarbeitungs-/Reflexionsphase

**Name:** Yannic Dykow
**Matrikelnummer:** 3220807
**Datum:** 11. Juni 2025

---

# 1. Untersuchung der Umsetzung objektorientierter Konzepte in Python

### 1.1 Klassenimplementierung in Python

Die Umsetzung der in Phase 1 modellierten Entity-Klassen in Python zeigt einige interessante Besonderheiten der Sprache. Im Gegensatz zu klassischen objektorientierten Sprachen wie Java benötigt Python keine expliziten Interface-Deklarationen oder strenge Zugriffsmodifikatoren.

Die Klasse `Pruefungsleistung` demonstriert die Python-typische Implementierung:

- Der Konstruktor `__init__` initialisiert Instanzvariablen direkt ohne vorherige Deklaration
- Methoden wie `ist_bestanden()` implementieren die Geschäftslogik kompakt
- Die Serialisierung mittels `to_dict()` und `from_dict()` zeigt das Duck-Typing-Prinzip

Ein wesentlicher Unterschied zu anderen OO-Sprachen ist die fehlende Kapselung: Alle Attribute sind standardmäßig public.

### 1.2 Python-spezifische Anpassungen

Drei wesentliche Anpassungen waren notwendig:

1. **Type Hints:** Zur besseren Dokumentation und IDE-Unterstützung wurden Type Annotations eingeführt: `def berechne_notenschnitt(self) -> float:`
2. **Klassenmethoden für Factory-Pattern:** Die `from_dict()` Methoden nutzen das `@classmethod` Decorator für elegante Deserialisierung
3. **None-Checks statt Null-Pointer:** Python's None erfordert explizite Prüfungen, was zu defensiverem Code führt

### 1.3 Kritische Reflexion der Modellierung

Die ursprüngliche Modellierung aus Phase 1 erwies sich als grundsätzlich solide, jedoch zeigten sich bei der Implementierung zwei Schwachstellen:

1. **Fehlende Statusverwaltung:** Module können verschiedene Zustände haben (geplant, angemeldet, in Bearbeitung, abgeschlossen), was im ursprünglichen Modell nur implizit über das Vorhandensein einer Prüfungsleistung abgebildet wurde.

2. **Zeitliche Aspekte:** Die aktuelle Implementierung ermöglicht keine Historisierung von Prüfungsversuchen - ein realistisches System müsste mehrere Versuche pro Modul unterstützen.

# 2. Gesamtarchitektur des Dashboard-Prototypen

## 2.1 Architektonische Entscheidung: Streamlit

Die Entscheidung, von tkinter zu Streamlit zu wechseln, basiert auf mehreren Überlegungen:

- **Rapid Prototyping:** Streamlit ermöglicht deutlich schnellere UI-Entwicklung
- **Moderne Visualisierungen:** Integration von Plotly für ansprechende Diagramme
- **Reaktives Paradigma:** Automatisches Re-rendering bei Zustandsänderungen

Diese Entscheidung beeinflusst die Gesamtarchitektur erheblich, da Streamlit ein deklaratives UI-Paradigma verwendet.

## 2.2 Erweiterte Klassenarchitektur

Zusätzlich zu den Entity-Klassen wurden folgende Komponenten identifiziert:

**Controller-Klassen:**

- `DataManager`: Verwaltung der Persistierung (JSON-Serialisierung)
- `VisualizationController`: Erstellung der Plotly-Diagramme

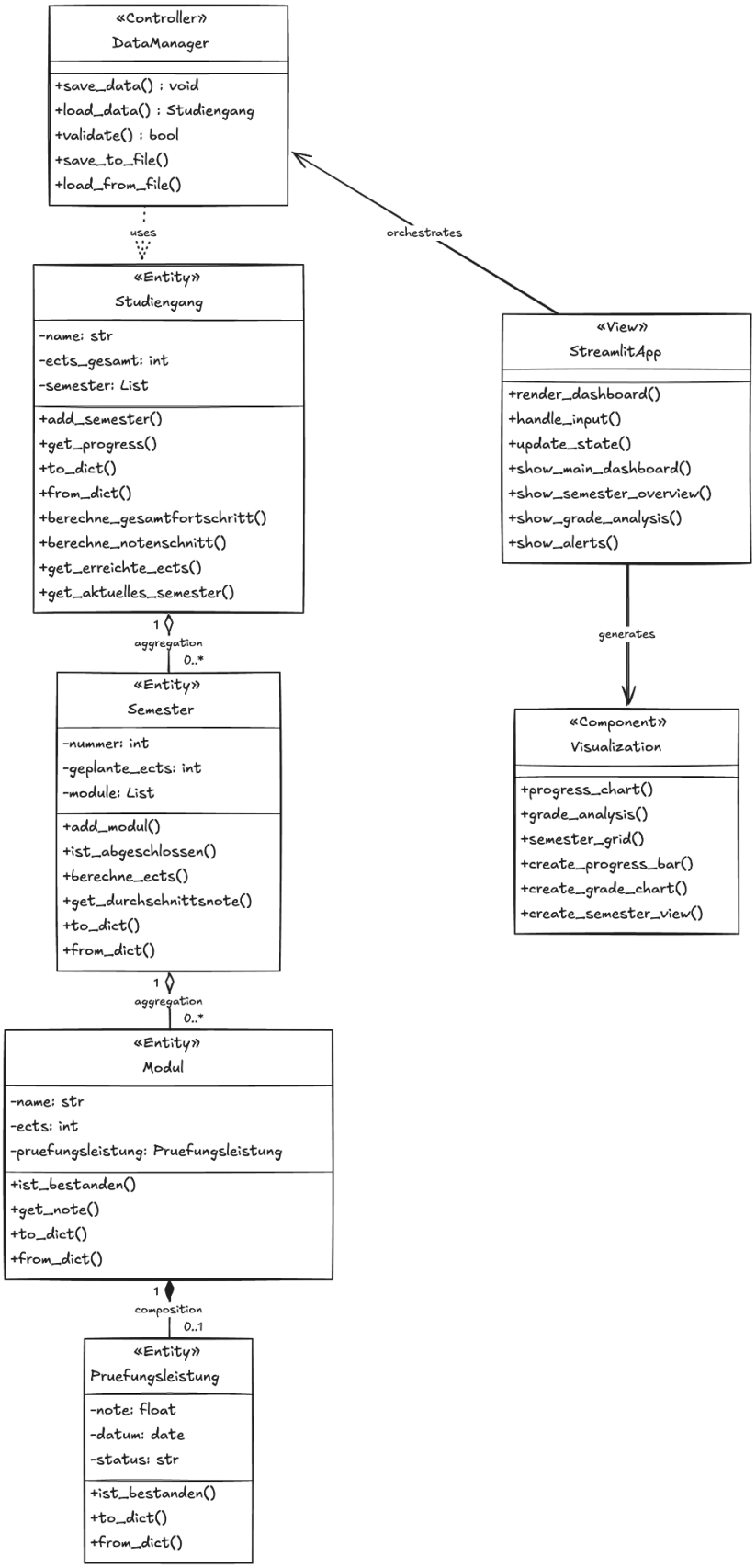
**View-Komponenten (in gui.py):**

- Dashboard-Hauptansicht
- Semester-Detailansicht
- Modul-Verwaltungsformulare

**Utility-Module:**

- `data.py`: (Beispiel)Daten-Generator
- `main.py`: Entry-Point mit Dependency-Check

## 2.3 UML-Klassendiagramm der Gesamtarchitektur



### 2.4 Begründung der Architekturentscheidungen

Die Trennung in Model-View-Controller (MVC) erfolgte bewusst, um:

1. Testbarkeit zu erhöhen - Entity-Klassen sind unabhängig vom UI testbar
2. Wartbarkeit zu verbessern - Änderungen am UI beeinflussen nicht die Geschäftslogik
3. Erweiterbarkeit zu ermöglichen - Neue Features können modular hinzugefügt werden

Die Entscheidung gegen ein komplexeres Repository-Pattern oder eine Service-Layer wurde getroffen, da für einen Prototyp die direkte JSON-Persistierung ausreichend ist. In einer Produktivumgebung würde hier eine Datenbankanbindung mit entsprechenden Abstraktionsschichten implementiert werden.

### 2.5 Streamlit-spezifische Überlegungen

Streamlit's Session State Management erforderte besondere Aufmerksamkeit:

- Zustandsverwaltung über `st.session_state` für persistente Daten zwischen Reloads
- Formulare mit eindeutigen Keys zur Vermeidung von Widget-Konflikten
- Strategischer Einsatz von `st.rerun()` für UI-Updates

## 3. Fazit und Ausblick

Die erweiterte Architektur mit Streamlit als UI-Framework bietet eine solide Basis für den Dashboard-Prototyp. Die klare Trennung von Geschäftslogik und Präsentation ermöglicht zukünftige Erweiterungen wie:

- REST-API für mobile Clients
- Datenbankanbindung statt JSON-Persistierung
- Erweiterte Analysefunktionen (Prognosen, Trends)