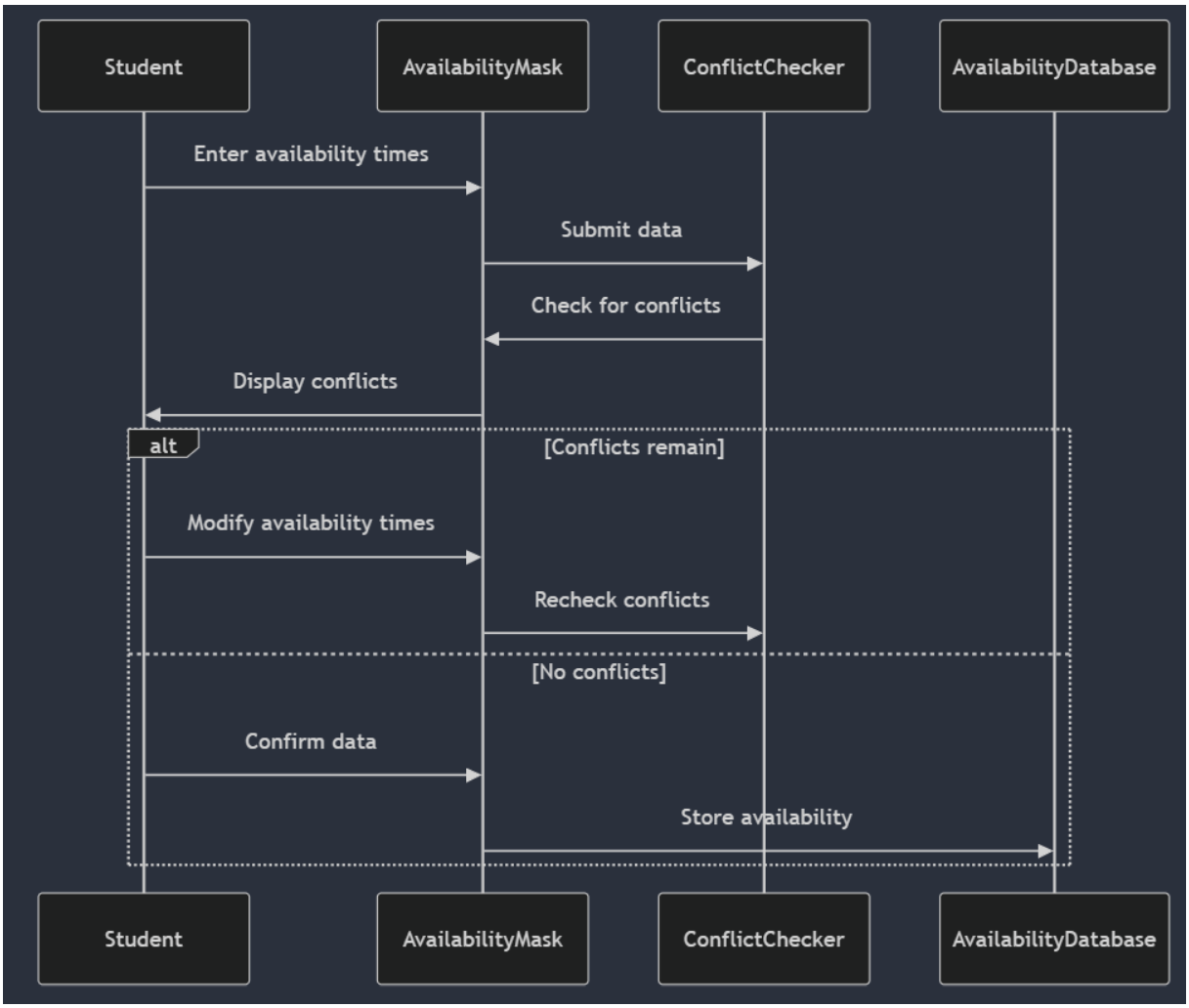
Aufgabe 1: Sequence Diagram

Problemstellung:
Ein Student gibt seine Verfügbarkeiten in eine Maske ein. Nach der Eingabe prüft ein Konfliktprüfer, ob es Überschneidungen mit Kursen gibt, für die der Student sich registriert hat. Falls Konflikte bestehen, wird dieser Prozess wiederholt. Sind keine Konflikte mehr vorhanden, bestätigt der Student die Eingabe, und die Daten werden in der Verfügbarkeitsdatenbank gespeichert.

Lösung:



Aufgabe 2: Interface Design

a) Design by Contract (DbC)

Vorbedingungen (Preconditions):

- courseName darf nicht null sein.
- Der Kurs darf noch nicht in der Liste enthalten sein.

context CourseManager::addCourse(courseName: String)

pre: courseName <> null and not courses->includes(courseName)

Nachbedingungen (Postconditions):

- Der Kurs wird zur Liste hinzugefügt.
- Die Größe der Liste erhöht sich um 1.

post: courses->includes(courseName) and courses->size() = courses@pre->size() + 1

Invarianten:

- Alle Einträge in der Kursliste dürfen nicht null sein.
- Die Liste enthält keine Duplikate.

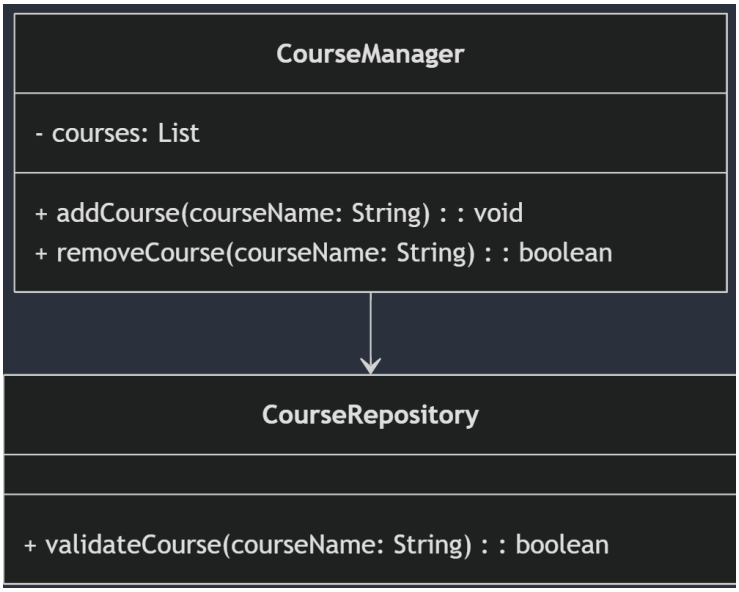
context CourseManager

inv: courses->forAll(c | c <> null) and courses->isUnique(c)

b) Analyse der SOLID-Prinzipien

1. **Single Responsibility Principle (SRP):**
   Problem:
   - Die Klasse CourseManager ist für die Verwaltung der Liste und die Validierung der Eingaben verantwortlich.
     Lösung:
   - Einführung einer neuen Klasse CourseValidator, die sich um die Validierung kümmert.
2. **Open/Closed Principle (OCP):**
   Problem:
   - Änderungen an der Validierungslogik erfordern Änderungen an der Methode addCourse.
     Lösung:
   - Verwende die neue Klasse CourseValidator, die mit einer modularen Struktur Änderungen zulässt.
3. **Liskov Substitution Principle (LSP):**
   - **Kein Problem:** Da es keine Vererbung in der aktuellen Struktur gibt.
4. **Interface Segregation Principle (ISP):**
   - **Kein Problem:** Die Klasse erfüllt spezifische Aufgaben ohne zu große Schnittstellen.
5. **Dependency Inversion Principle (DIP):**
   Problem:
   - CourseManager ist direkt von der Implementierung der ArrayList abhängig.
     Lösung:
   - Verwende die List-Schnittstelle anstelle der konkreten Klasse.

Klassendiagramm für die Lösung:



1. **CourseManager**:
    - Verwaltet die Liste der Kurse.
    - Enthält Methoden für das Hinzufügen (addCourse) und Entfernen (removeCourse) von Kursen.
2. **CourseRepository**:
    - Eine separate Klasse, die sich ausschließlich um die Validierung von Kursen kümmert.
    - Ermöglicht die Einhaltung des **Single Responsibility Principle (SRP)**.
3. **Beziehung**:
    - CourseManager verwendet die Dienste von CourseRepository für die Kursvalidierung.

Mit diesem Ansatz wird die Trennung der Verantwortlichkeiten (SRP) gewährleistet und die Abhängigkeiten des Systems modular gestaltet. Sollten sich die Validierungsregeln ändern, müssen nur Anpassungen in CourseRepository vorgenommen werden.

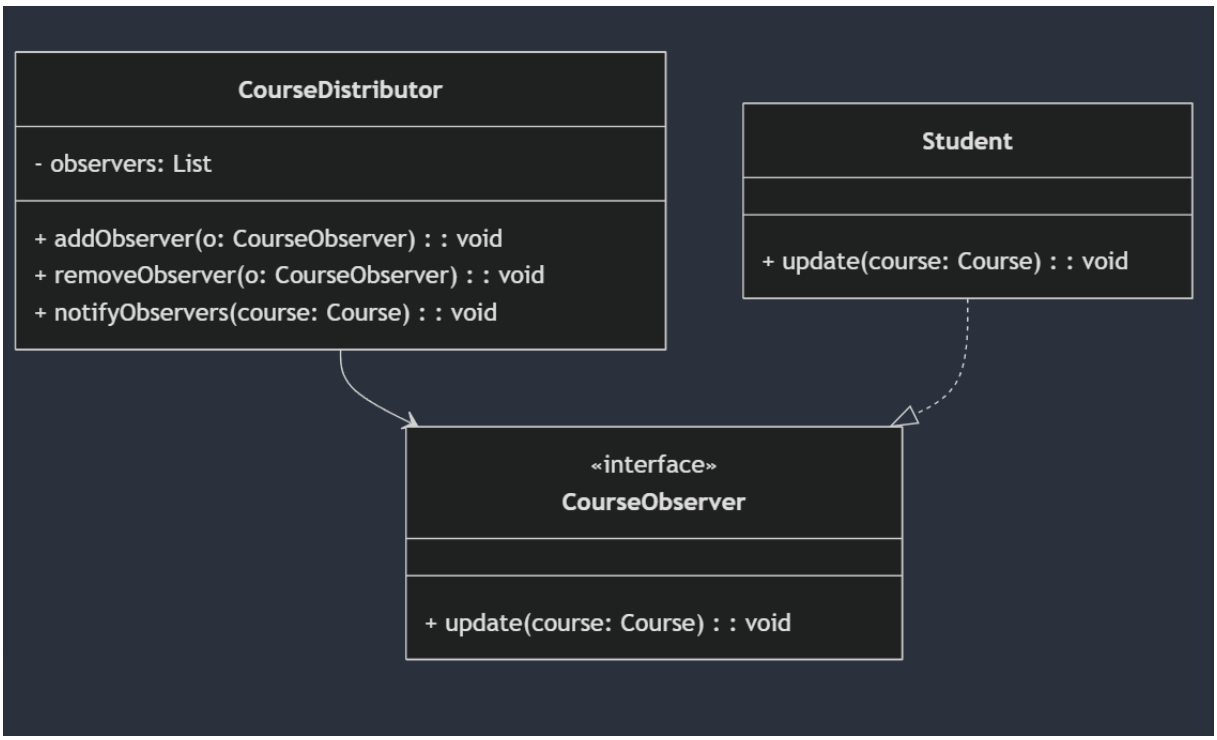
Aufgabe 3: Observer Pattern

Das Observer Pattern ermöglicht eine flexible Benachrichtigungsarchitektur für das Kursverteilungssystem. Die zentrale Idee besteht darin, eine lose Kopplung zwischen dem Kursverteiler und den zu benachrichtigenden Studenten zu etablieren. Über eine gemeinsame Schnittstellendefinition CourseObserver können beliebig viele Studenten als Beobachter registriert werden. Der CourseDistributor verwaltet eine dynamische Liste dieser Beobachter und kann sie bei Änderungen gezielt benachrichtigen.

Die Implementierung sieht vor, dass jeder Student die update()-Methode implementiert, die automatisch aufgerufen wird, wenn sich Informationen zu einem Kurs ändern. Der Kursverteiler seinerseits bietet Methoden zum Hinzufügen und Entfernen von Beobachtern sowie eine zentrale notifyObservers()-Methode. Diese Herangehensweise erlaubt es, neue Benachrichtigungsempfänger ohne Veränderung der bestehenden Systemarchitektur hinzuzufügen.

Die Vorteile dieses Ansatzes liegen in der hohen Flexibilität und Erweiterbarkeit. Studenten können dynamisch dem Benachrichtigungssystem beitreten oder es verlassen, ohne dass Änderungen an der Kursverteilungslogik erforderlich sind. Die strikte Trennung der Verantwortlichkeiten fördert eine modulare und wartbare Softwarearchitektur.

Darstellung:



Aufgabe 4: Strategy Pattern

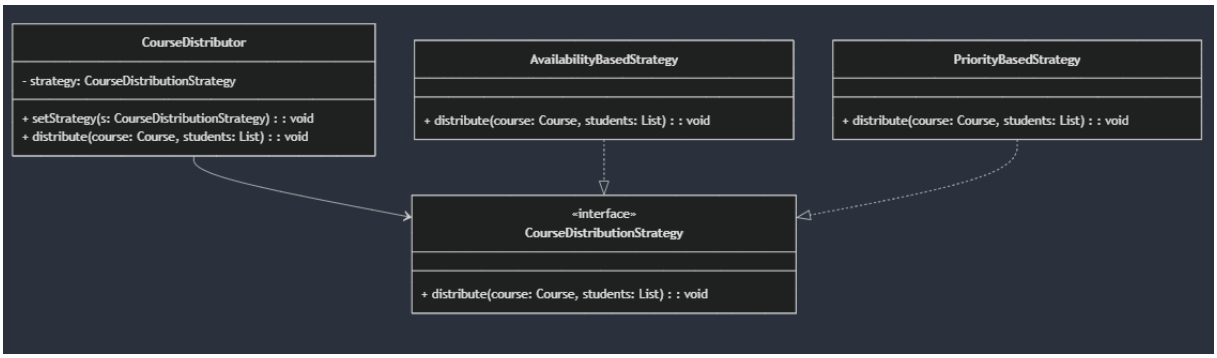
Das Strategy Pattern bietet eine elegante Lösung zur Implementierung verschiedener Kursverteilungsparadigmen. Kern der Lösung ist eine zentrale Schnittstellendefinition CourseDistributionStrategy, die eine einheitliche Verteilungsmethode vorgibt. Konkrete Strategien wie AvailabilityBasedStrategy und PriorityBasedStrategy implementieren diese Schnittstelle und realisieren jeweils ihre spezifische Verteilungslogik.

Der CourseDistributor erhält die Fähigkeit, Verteilungsstrategien zur Laufzeit zu wechseln. Dies geschieht über eine setStrategy()-Methode, die es Dozenten ermöglicht, dynamisch zwischen verschiedenen Verteilungsparadigmen zu wechseln. Die Course-Klasse kann dabei eine aktuelle Verteilungsstrategie speichern und bei Bedarf anpassen.

Die Architektur erlaubt es, neue Verteilungsstrategien einfach durch Hinzufügen neuer Strategieklassen zu implementieren, ohne bestehenden Code modifizieren zu müssen. Zukünftige Verteilungsansätze können so nahtlos in das System integriert werden. Die Kapselung der Verteilungslogik in separaten Strategieklassen fördert die Wartbarkeit und Erweiterbarkeit des Gesamtsystems.

Durch diese Herangehensweise wird eine hohe Flexibilität erreicht: Neue Verteilungsparadigmen können entwickelt und integriert werden, ohne die grundlegende Systemarchitektur zu verändern. Dozenten erhalten die Möglichkeit, Verteilungsstrategien schnell und unkompliziert anzupassen.

Darstellung:



Alle Diagramme wurden mit memaid.live erstellt.