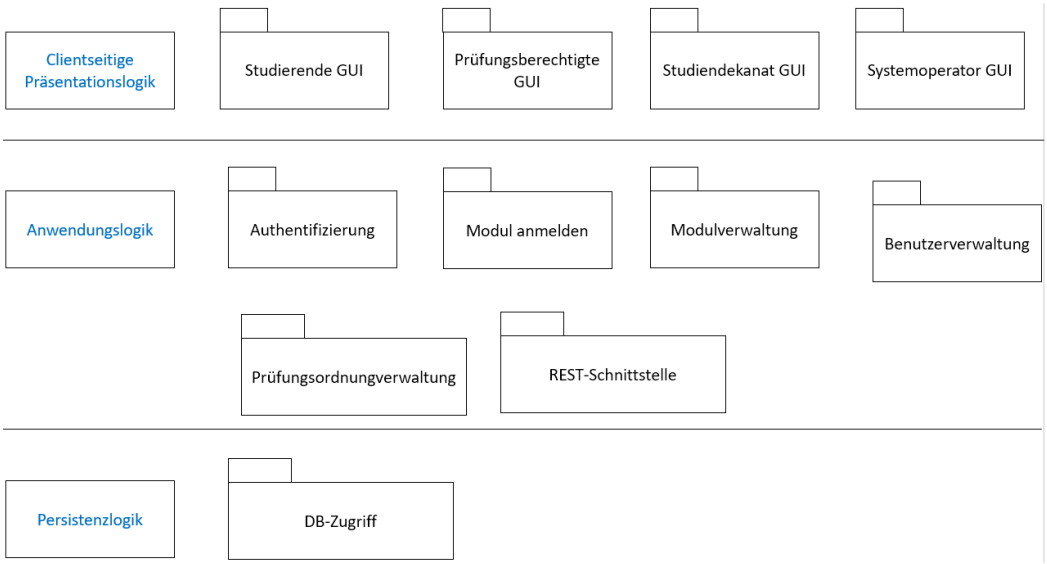
Aufgabe 8.1: Design des Modulbelegungsmanagementsystems ModMS - Pflichtaufgabe
Für das Modulbelegungsmanagementsystem aus den Übungsblättern 4, 5 und 6 soll ein Design durchgeführt werden.

(a) Entwerfen Sie ein physikalisches Modell für das System, und zwar in Form eines Deployment-Diagramms mit den notwendigen Hardware-Einheiten. Geben Sie die bei der Kommunikation verwendeten Protokolle an, soweit Ihnen diese bekannt sind.

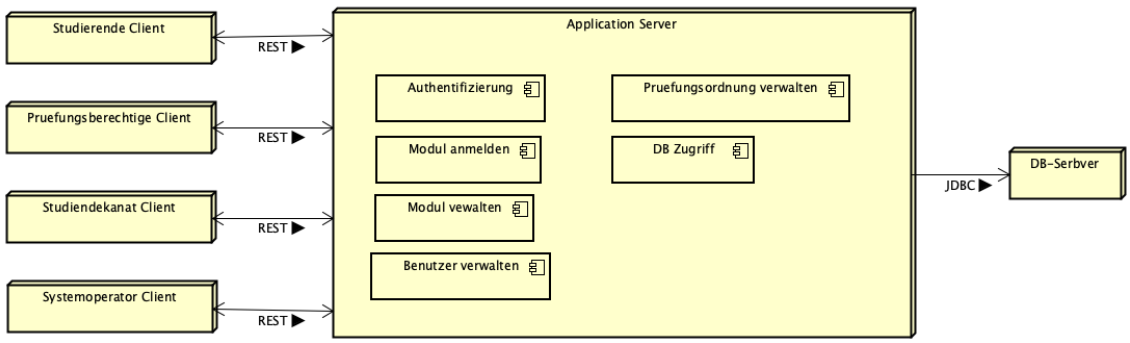
(b) Entwickeln Sie ein Modell für die logische Struktur des Systems. Zerlegen Sie das System hierzu in logische Teilsysteme und identifizieren Sie die fehlenden technischen Komponenten. Legen Sie exemplarisch (1) die klar definierte Verantwortung und (2) eine vollständig beschriebene Schnittstelle für ein Teilsystem fest. Ordnen Sie das fachliche Klassendiagramm aus Übungsblatt 6 (Aufgabe 3) den logischen Komponenten zu.

(c) Verteilen Sie die logischen Komponenten auf die jeweiligen Knoten des physikalischen Modells.

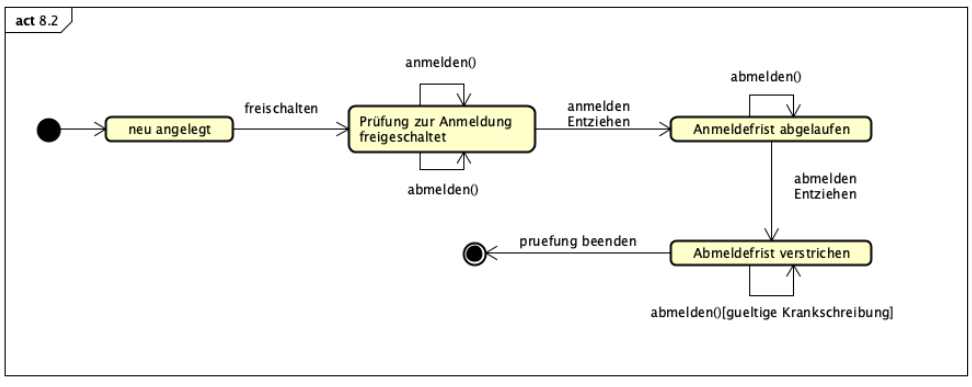
b.)



a.) und c.)



Aufgabe 8.2: Zustandsdiagramm für Modulbelegungsmanagementsystem ModMS
Erstellen Sie ein Zustandsdiagramm für die Klasse `Prüfung` aus Übungsblatt 6. Jedes Semester wird eine Prüfung (z.B. für SE1 im SS22) neu angelegt, für die sich dann Studierende an- und abmelden können, sobald die Prüfung zur Anmeldung freigeschaltet ist. Nach Ablauf der Anmeldefrist sind für einen bestimmten Zeitraum noch Abmeldungen möglich, bis auch die Abmeldefrist verstrichen ist und nur noch Krankmeldungen zulässig sind. Bestimmten Sie hierzu die prinzipiell möglichen Zustände mit den jeweils erlaubten Operationen/Ereignissen.
Bitte beachten Sie, dass es sich hierbei um die generelle An-/Abmeldung zu einer Prüfung handelt und nicht um die An-/Abmeldung einer/s bestimmten Studierenden. (Pro Anmeldung eines/r Studierenden würde ein entsprechendes Objekt der Klasse `Prüfungsversuch` erzeugt.)



Aufgabe 8.3: Singleton-Pattern

Bei gewissen Klassen ist es wichtig, dass es von diesen nur genau **ein einziges** Objekt (Exemplar) geben darf. Oft sind dies Klassen, die zentrale Systemdienste zur Verfügung stellen, zum Beispiel:

- Container-Klassen zur Verwaltung von Objekten soll es nur einmal geben.
- Eine Klasse, die bei einem Brettspiel die Steuerung des Spielablaufs übernimmt (oder die Klasse, die die Spielsituation verwaltet).

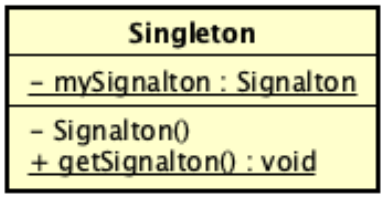
Gäbe es im Laufzeitsystem mehrere Instanzen von solchen Klassen, dann wäre es schwierig deren Konsistenz sicherzustellen.

Überlegen Sie sich ein Entwurfsmuster für eine Klasse **Singleton**, die nur **ein einziges** Objekt besitzen darf.

- Stellen Sie sicher, dass nur ein einziges Objekt dieser Klasse erzeugt werden kann.
- Realisieren Sie einen zentralen Zugriffspunkt, d.h. über eine Methode soll entweder das Objekt erzeugt werden (falls es noch keins gibt) oder eine Referenz auf das bereits erzeugte Objekt geliefert werden.

a) Zeichnen Sie ein entsprechendes UML-Modell.
b) Implementieren Sie Ihre Lösung in Java.
c) Durch Singleton kann ja nur genau ein Objekt erzeugt werden. Warum arbeitet man nicht direkt mit einer statischen Klasse (mit ausschließlich mit statischen Methoden) und verzichtet auf das Singleton?
d) Welche Nachteile kann die Verwendung einer Singleton-Klasse mit sich bringen?

a.)



b.)

```
public class Singleton {
private static Singleton obj = null;
private Singleton() { }
public static Singleton getSingleton (){
    if (obj == null)
        obj = new Singleton();
    return  obj;
    }
}
```

c.)

- Vererbung von Methoden funktionieren nicht und können auch nicht überschrieben werden, also Polymorphie nicht möglich

d.)

Vorteile

- einfach zu implementieren
- vermeidet globale Variable: lazy loading, Zugriffskontrolle, ...

Nachteile

- OO-Variante globaler Variablen
  - versteckte Abhängigkeiten: nicht in API der Objekte sichtbar
  - schlecht zu testen
- Änderung des Typs wg. statischer Methode nicht möglich:
  - Name der Singleton-Klasse = Typ des erzeugten Objekts
    => Factory-Patterns liefern mehr Flexibilität
- im Kontext verteilter / multi-threaded Anwendungen gibt es Probleme:
  - getInstance() muss thread-safe sein, damit wirklich nur 1 Objekt erzeugt wird
  - Singleton kann dann ein Performance-Flaschenhals sein