Universität Augsburg
Institut für Informatik

Übung zur Vorlesung Informatik 2 SoSe 2024
Fakultät für Angewandte Informatik
Institut für Informatik
Prof R. Lorenz, L. Petrak, J. Linne, V. Le Claire, P. Schalk 21.06.2024

Lösungsvorschlag zu Übungsblatt 8

Abgabe: 01.07.2024, 10:00 Uhr (im Digicampus via VIPS: .java-Dateien für Code, .uxf für UML, .pdf für alles andere)

- Dieses Übungsblatt muss im Team abgegeben werden (Einzelabgaben sind nicht erlaubt!).
- Die **Zeitangaben** geben zur Orientierung an, wie viel Zeit für eine Aufgabe später in der Klausur vorgesehen wäre; gehen Sie davon aus, dass Sie zum jetzigen Zeitpunkt wesentlich länger brauchen und die angegebene Zeit erst nach ausreichender Übung erreichen.

* leichte Aufgabe / ** mittelschwere Aufgabe / *** schwere Aufgabe

Allgemeine Hinweise zur Erstellung und Abgabe von UML-Diagrammen:

- UML-Diagramme sind mit dem Editor **UMLet**[1] zu erstellen (es darf auch die webbasierte Variante **UMLetino**[2] verwendet werden). Damit die Leserichtung von Assoziationen beim PDF-Export aus UMLet erhalten bleibt, müssen Sie eine Schriftart installieren, welche die verwendeten Zeichen beinhaltet, wie z.B. `https://fonts2u.com/download/free-sans.family`. Exportieren Sie die enthaltene .zip-Datei und tragen Sie den absoluten Pfad zu den Dateien in UMLet unter *File* → *Options* → *Optional font to embedd in PDF* für alle vier Varianten ein, verwenden Sie die `Oblique`-Dateien für *italic text* und *bold+italic*.
- Die (graphischen) Syntaxvorgaben aus der Vorlesung sind einzuhalten.
- Modellieren Sie Attribute, Konstruktoren und Operationen im Stil der Vorlesung (z.B. im Hinblick auf die Benennung).
- Erfinden Sie keine Daten oder Operationen hinzu, die nicht der jeweiligen Beschreibung entnommen werden können.
- Wenn Sie eine API-Klasse bzw. -Schnittstelle als Eingabeparameter oder Rückgabetyp verwenden, reicht die Angabe als `java.package.Klasse` ohne Modellierung als eigene Klasse bzw. Schnittstelle.
- Wenn Sie Beziehungen zu API-Klassen bzw. Schnittstellen wie Vererbung oder Implementierung benötigen, modellieren Sie die Klassen bzw. Schnittstellen mit derselben Namensgebung `java.package.Klasse`.
- Operationen und Attribute von API-Klassen und Assoziationen zwischen API-Klassen werden nur dann modelliert, wenn sie für die Funktionalität der modellierten Anwendung notwendig sind oder wenn die Aufgabenstellung es explizit verlangt.
- UML-Diagramme sind **sowohl** als UMLet-Datei (`*.uxf`) **als auch** als PDF abzugeben.

[1] `https://www.umlet.com`
[2] `http://www.umletino.com`

Aufgabe 29 ** *(UML-Klassendiagramm modellieren, 22 Minuten)*

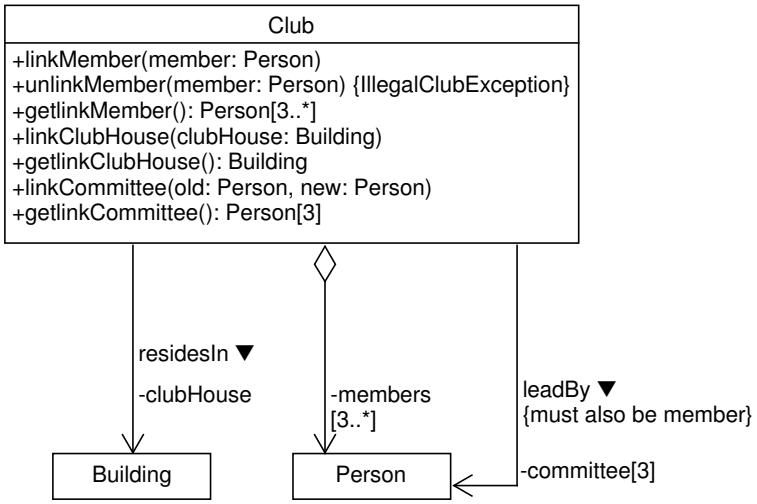
In dieser Aufgabe sollen Sie eigene UML-Klassen und deren Beziehungen untereinander modellieren. Für Ausnahmen dürfen Sie passende Java-API-Ausnahmen, in der Vorlesung eingeführte Ausnahmen oder auch eigene neue Ausnahmen mit passendem Namen verwenden.

a) *(**, 12 Minuten)*
Betrachten Sie folgende Beschreibung eines Systems zur **Verwaltung von Vereinen**:

Ein Verein besteht aus mindestens 3 Personen. Eine Person kann auch in mehreren Vereinen Mitglied sein. Ein Verein hat immer genau drei Vorstände, diese müssen auch Mitglieder des Vereins sein. Jeder Verein hat ein Vereinshaus.

Repräsentieren Sie das beschriebene System durch ein in sich konsistentes und vollständiges UML-Klassendiagramm. Modellieren Sie Klassen für die beschriebenen Objekte **ohne** Attribute oder Konstruktoren. Geben Sie passende Standardverwaltungsoperationen an. Geben Sie für die Beziehungen auch Bezeichnungen und Leserichtungen an.

Lösung:

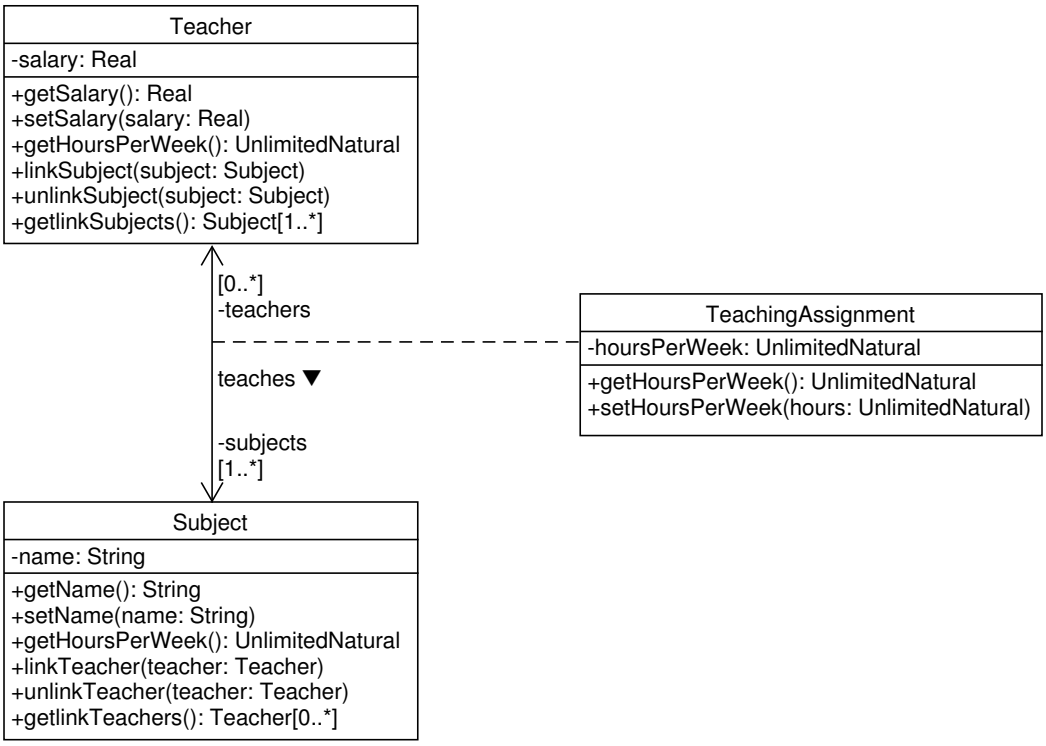


b) *(**, 10 Minuten)*
Betrachten Sie folgende Beschreibung eines Systems zur **Verwaltung von Lehrern und Schulfächern**:

Ein Lehrer unterrichtet mindestens ein Schulfach. Ein Lehrer bekommt einen Lohn, der reellwertig ist. Ein Schulfach hat einen Namen, der durch eine Zeichenkette ausgedrückt wird. Ein Schulfach kann von keinem, einem oder mehreren Lehrern unterrichtet werden. Ein Lehrer unterrichtet ein Schulfach immer eine bestimmte Anzahl von Stunden pro Woche. Es soll möglich sein, zu berechnen, wie viele Stunden ein Lehrer pro Woche insgesamt unterrichtet. Genauso soll berechnet werden können, wie viele Stunden ein Schulfach pro Woche unterrichetet wird.

Repräsentieren Sie das beschriebene System durch ein in sich konsistentes und vollständiges UML-Klassendiagramm. Verwenden Sie eine Assoziationsklasse, um das Unterrichten zu modellieren. Geben Sie dabei so detailliert wie möglich Klassen zur Verwaltung der beschriebenen Informationen / Daten mit Attributen und Beziehungen an. Geben Sie Standardverwaltungsoperationen für alle Klassen an, aber **keine** Konstruktoren.

Lösung:



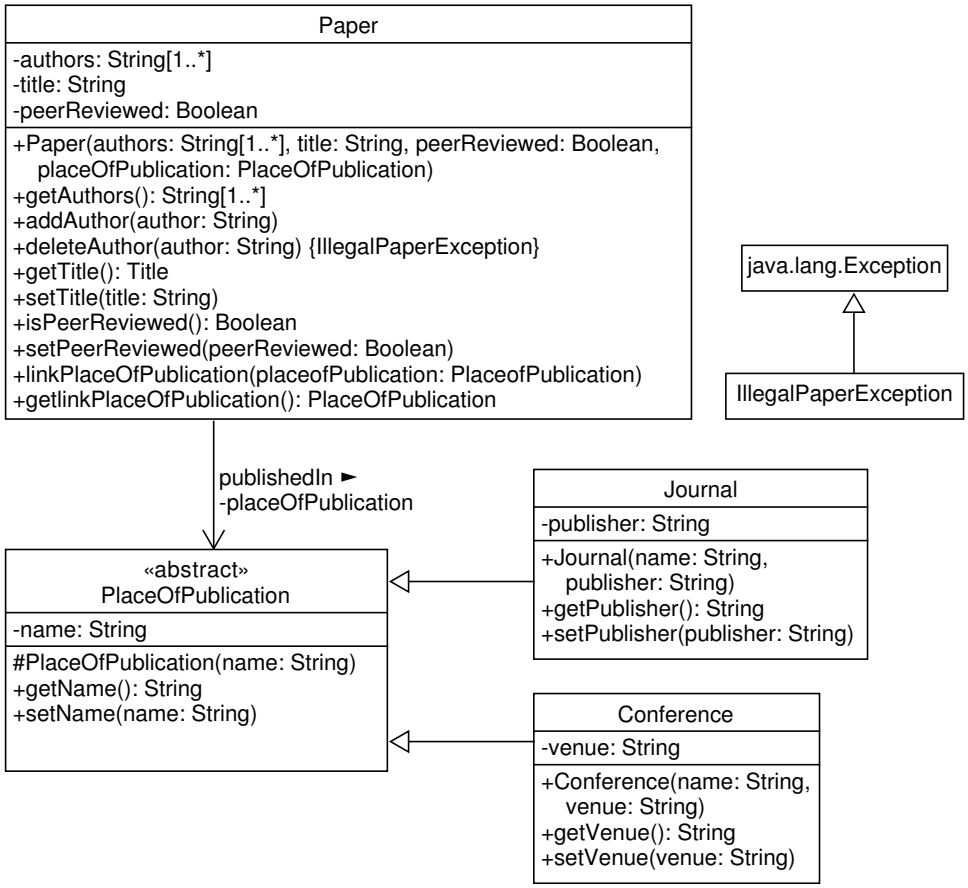
Aufgabe 30 ** *(UML-Klassendiagramm modellieren, 20 Minuten)*

In dieser Aufgabe sollen Sie eigene UML-Klassen und deren Beziehungen untereinander modellieren. Für Ausnahmen dürfen Sie passende Java-API-Ausnahmen, in der Vorlesung eingeführte Ausnahmen oder auch eigene neue Ausnahmen mit passendem Namen verwenden.

Betrachten Sie die folgende Beschreibung eines **Systems zur Verwaltung von wissenschaftlichen Papers**:
Ein Paper hat einen Titel sowie mindestens einen Autor. Ein Paper kann peer-reviewed sein oder nicht. Außerdem wird ein Paper in einem Medium veröffentlicht: Ein Paper kann bei einer Konferenz oder in einem Journal veröffentlicht werden. Ein Medium hat einen Namen. Ein Journal hat einen Verlag, während eine Konferenz einen Veranstaltungsort hat. Es gibt keine allgemeinen Medien, sondern bis jetzt nur Journals und Konferenzen. Es soll leicht möglich sein, weitere Arten von Medien in das System einzupflegen.

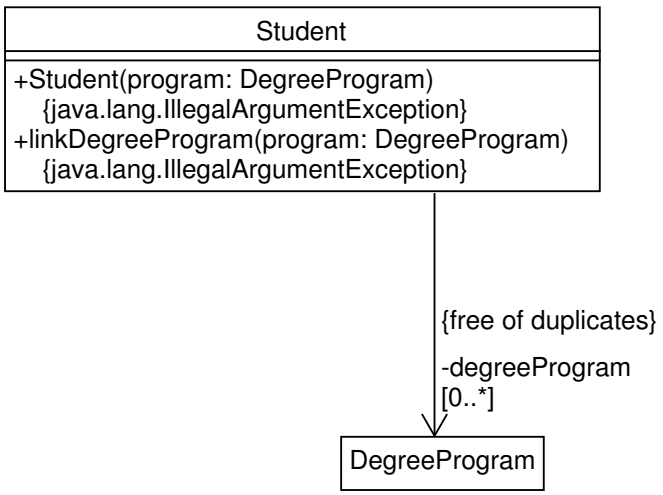
Repräsentieren Sie das beschriebene System durch ein in sich konsistentes und vollständiges UML-Klassendiagramm. Geben Sie dabei so detailliert wie möglich Klassen zur Verwaltung der beschriebenen Informationen / Daten mit Attributen und Beziehungen an. Geben Sie auch Konstruktoren und Standardverwaltungsoperationen an.

Lösung:



Aufgabe 31 ** *(UML-Klassendiagramm in Java implementieren, 16 Minuten)*

a) *(*, Alte Klausuraufgabe, 6 Minuten)*
Implementieren Sie die Klasse `Student` aus dem folgenden Klassendiagramm:



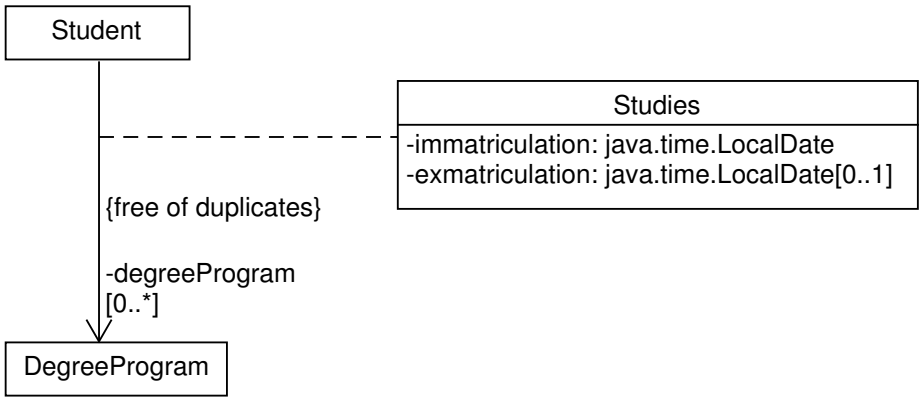
Implementieren Sie nur die im Modell enthaltenen Elemente der Klasse `Student`.
Lösung:

```
import java.util.Set;
import java.util.HashSet;

public class Student {

    private Set<DegreeProgram> degreeProgram = new HashSet<>();

    public Student(DegreeProgram program) {
        linkDegreeProgram(program);
    }

    public void linkDegreeProgram(DegreeProgram program) {
        if (!degreeProgram.add(program))
            throw new IllegalArgumentException("program already present.");
    }

}
```

Diskussion:

- Statt `HashSet` dürfte auch eine andere `Collection`-Implementierung gewählt werden. Beachten Sie, dass keine Duplikate in der `Collection` vorkommen dürfen.

b) *(**, 4 Minuten)*
Implementieren Sie das folgende Klassendiagramm:

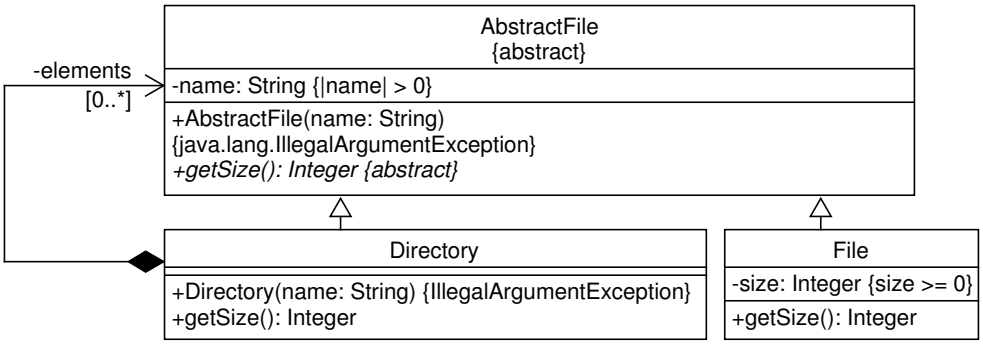


Implementieren Sie nur die im Modell enthaltenen Elemente.

Lösung:

```
import java.util.Set;
import java.util.HashSet;

public class Student {

    private Set<Studies> studies = new HashSet<>();

}

public class Studies {

    private java.time.LocalDate immatriculation;
    private java.time.LocalDate exmatriculation;
    private DegreeProgram degreeProgram;

}

public class DegreeProgram {

}
```

c) *(**, 6 Minuten)*
Implementieren Sie die Klasse `Directory` aus dem folgenden Klassendiagramm:



Hinweise:

- Die Größe eines Verzeichnisses oder einer Datei (`size`) repräsentiert hier den Speicherbedarf.
- Implementieren Sie nur die im Modell enthaltenen Elemente der Klasse `Directory`.

Lösung:

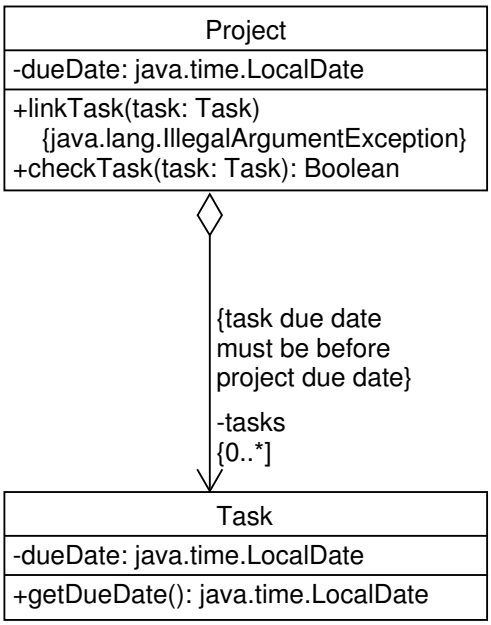
```
import java.util.ArrayList;
import java.util.List;

public class Directory extends AbstractFile {

    private List<AbstractFile> elements = new ArrayList<>();

    public Directory(String name) {
        super(name);
    }

    @Override
    public int getSize() {
        return elements.stream().mapToInt(AbstractFile::getSize).sum();
    }

}
```

Diskussion:

- Die Größe eines Verzeichnisses ist die Summe der Größen seiner Elemente.
- Polymorphismus: Aufruf von `getSize` für Objekte vom Typ `AbstractFile`.

Aufgabe 32 ** *(UML-Klassendiagramm in Java implementieren, 25 Minuten)*

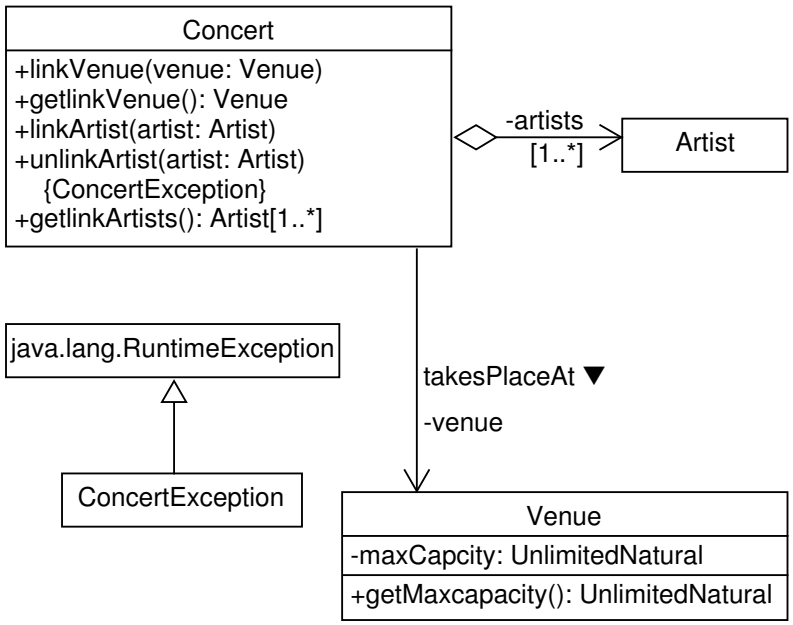
a) *(**, 3 Minuten)*
Implementieren Sie die Methoden `checkTask` und `linkTask` der Klasse `Project` basierend auf folgendem Klassendiagramm:



Lösung:

```
public void linkTask(Task task) {
    if (!checkTask(task))
        throw new IllegalArgumentException("Task due date must be before project due
        ↪  date");
    tasks.add(task);
}

public boolean checkTask(Task task) {
    return task.getDueDate().isBefore(dueDate);
}
```

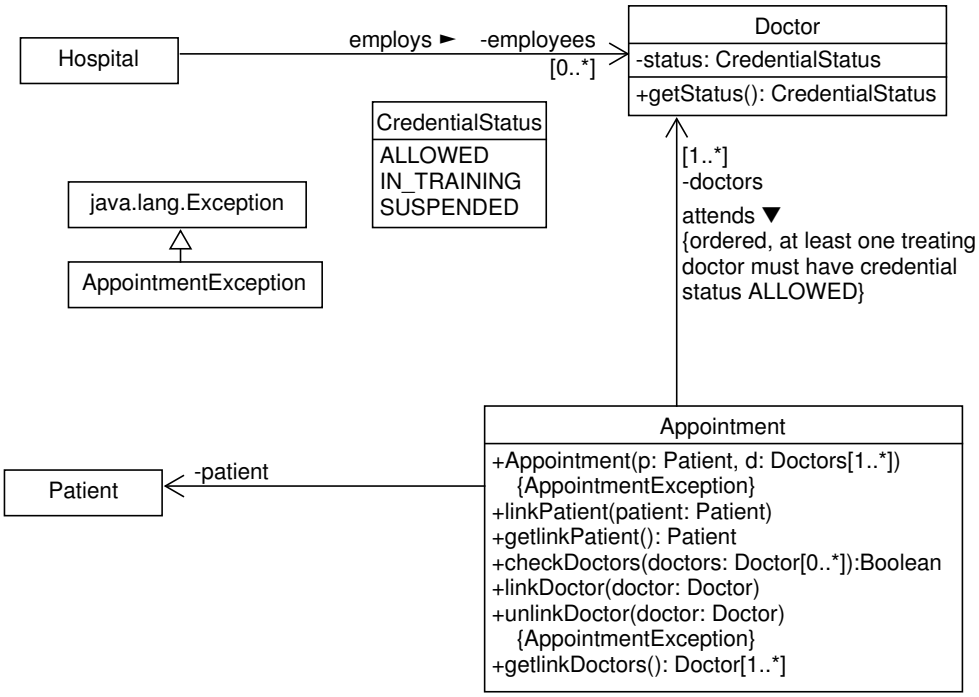
b) *(**, 7 Minuten)*
Implementieren Sie die Klasse `Concert` basierend auf folgendem Klassendiagramm:



Lösung:

```
import java.util.ArrayList;

public class Concert {
    private Venue venue;
    private ArrayList<Artist> artists;

    public void linkArtist(Artist artist) {
        artists.add(artist);
    }

    public void unlinkArtist(Artist artist) {
        if (artists.size() == 1) {
            throw new ConcertException("Concert must have at least one artist");
        }
        artists.remove(artist);
    }

    public ArrayList<Artist> getlinkArtist() {
        return artists;
    }

    public void linkVenue(Venue venue) {
        this.venue = venue;
    }

    public Venue getlinkVenue() {
        return venue;
    }
}
```

c) *(***, 15 Minuten)*
Implementieren Sie die Klasse `Appointment` aus dem folgenden Klassendiagramm:



Lösung:

```
import java.util.ArrayList;
import java.util.Collection;

public class Appointment {
    private Patient patient;
    private ArrayList<Doctor> doctors = new ArrayList<>();

    public Appointment(Patient patient, Collection<Doctor> doctors) throws
    ↪  AppointmentException {
        linkPatient(patient);
        if(!checkDoctors(doctors))
            throw new AppointmentException("Appointment must have at least one doctor with
            ↪  credential status ALLOWED");
        for (Doctor d: doctors)
            linkDoctor(d);
    }

    public void linkPatient(Patient patient) {
        this.patient = patient;
    }

    public Patient getlinkPatient() {
        return patient;
    }

    private static boolean checkDoctors(Collection<Doctor> doctors) {
        if (doctors.isEmpty())
            return false;
        for (Doctor d : doctors) {
            if (d.getStatus().equals(CredentialStatus.ALLOWED))
                return true;
        }
        return false;
    }
```

```
    public void linkDoctor(Doctor doctor) {
        doctors.add(doctor);
    }

    public void unlinkDoctor(Doctor doctor) throws AppointmentException {
        ArrayList<Doctor> copy = (ArrayList<Doctor>) doctors.clone();
        copy.remove(doctor);
        if (!checkDoctors(copy))
            throw new AppointmentException("Appointment must have at least one doctor with
            ↪  credential status ALLOWED");
        doctors.remove(doctor);
    }

    public Collection<Doctor> getlinkDoctors() {
        return doctors;
    }
}
```

Diskussion:

- Die Überprüfung auf `isEmpty()` in `checkDoctor` ist nicht notwendig, weil der zweite Teil auch `false` zurückgibt, wenn `doctors` leer ist.
- Es ist möglich, bei `unlinkDoctor` wie hier zuerst eine Kopie zu erzeugen, den übergebenen Doktor daraus zu entfernen und diese Liste zu überprüfen, bevor man die Entfernung auch in der eigentlichen Liste vornimmt. Stattdessen könnte man auch den Doktor direkt aus `doctors` entfernen, `doctors` überprüfen und im Fehlerfall den Doktor wieder hinzufügen.