Assignment 1 – UML

Provide class examples of the following relationships.

I have included a short explanation of assumptions made for each example to justify the design. Also, I may have mixed knowledge of this course with some previous knowledge (e.g. I don't think we've learned about getters and setters or Java specific datatypes like LocalDate/Period yet) – I hope that is not an issue. As proof that I did my research and it was not AI created ideas, I add screenshots (see Figure 1 & Figure 2) with sources and self-made pseudo-code examples for selected methods/functions (see Figure 5).

Date-time types in Java & SQL	Legacy class	Modern class	SQL standard data type
Moment in UTC	java.util. ~~Date~~ java.sql. ~~Timestamp~~	java.time. Instant	TIMESTAMP WITH TIME ZONE
Moment with offset-from-UTC (hours-minutes-seconds)	(lacking)	java.time. OffsetDateTime	TIMESTAMP WITH TIME ZONE
Moment with time zone (`Continent/Region`)	java.util. ~~GregorianCalendar~~ javax.xml.datatype. ~~XMLGregorianCalendar~~	java.time. ZonedDateTime	TIMESTAMP WITH TIME ZONE
Date & Time-of-day (no offset, no zone) Not a moment	(lacking)	java.time. LocalDateTime	TIMESTAMP WITHOUT TIME ZONE
Date only (no offset, no zone)	java.sql. ~~Date~~	java.time. LocalDate	DATE
Time-of-day only (no offset, no zone)	java.sql. ~~Time~~	java.time. LocalTime	TIME WITHOUT TIME ZONE
Time-of-day with offset (impractical, not used)	(lacking)	java.time. OffsetTime	TIME WITH TIME ZONE

JDBC 4.2

Figure 1: Research result on what datatype to use for date representation (https://stackoverflow.com/questions/28730136/should-i-use-java-util-date-or-switch-to-java-time-localdate).

ofMonths

```
public static Period ofMonths(int months)
```

Obtains a `Period` representing a number of months.

The resulting period will have the specified months. The years and days units will be zero.

Parameters:

`months - the number of months, positive or negative`

Returns:

`the period of months, not null`

Figure 2: Research result on what datatype to use for date interval representation & calculation (https://docs.oracle.com/javase/8/docs/api/java/time/Period.html).

One-to-One relationship

I have decided to design the relationship between *Patient* and *MedicalRecord*. I suppose that if handled strictly/more modern, a new *Patient* instantiation usually happens before their first *MedicalRecord*. So, if we include that thought it could also be a One-to-Zero or One relationship. But in this scenario, the stakeholders are old-school and first take in all the information on paper and then create the *Patient* and *MedicalRecord* simultaneously. Therefore, one *Person* has one *MedicalRecord*. Furthermore, it is a composition, because if a *Patient* is removed from the system, their *MedicalRecord* is also deleted.

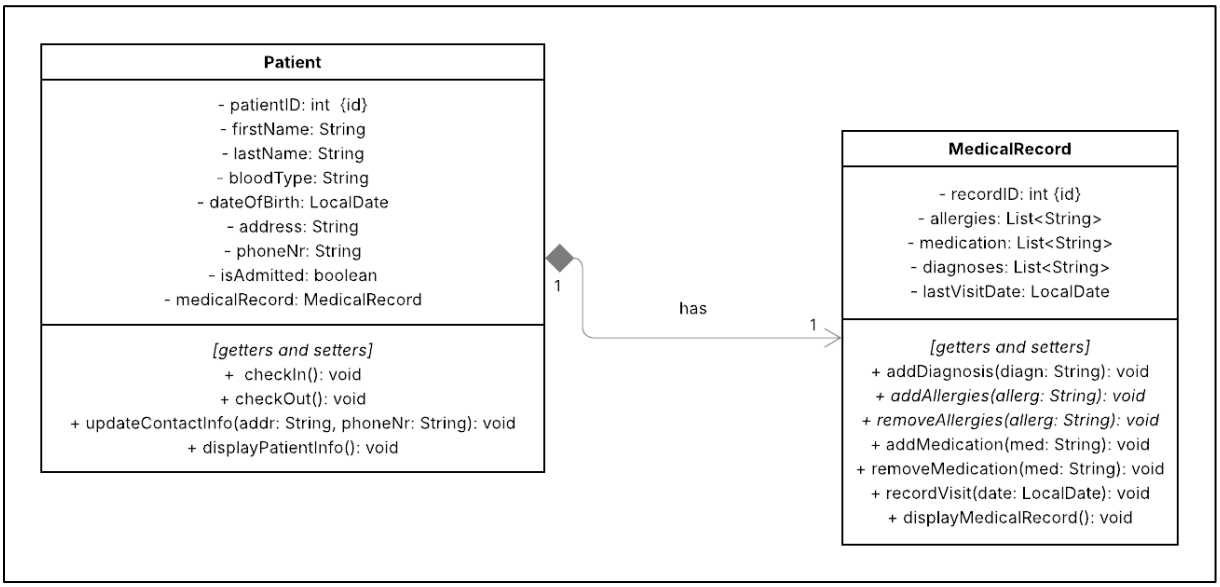

Figure 3: UML class-diagram for the chosen One-To-One relationship.

I suppose the attributes should be self-explanatory for both classes. For *Patient*, the checkIn and -Out methods update the isAdmitted status. I wouldn't update this over a setter, because they are meaningful domain-actions – not simple updates. The others should be self-explanatory. For *MedicalRecord*, Allergies and Medication can be removed as those can change over time. The existence of the method *recordVisit* is justified in the same way as checkIn and -Out.

Aggregation relationship

I have chosen an aggregation relationship between *Airline* and *Airplane*. In the chosen scenario, an *Airline* can have no or many instances of *Airplane*. I consider it an aggregation because the *Airplane* can still exist, even if the *Airline* does not (e.g. in some sort of hangar/storage or as part of a different *Airline* instance).

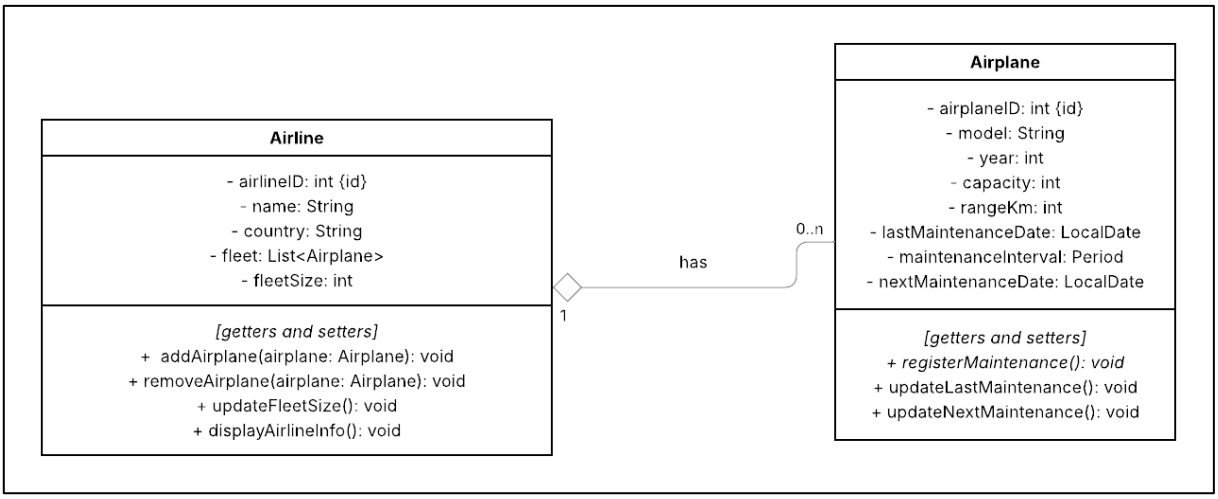

Figure 4: UML class-diagram for the chosen aggregation relationship.

The existence of the methods can be justified in the same way as in exercise 1. For proof that I designed this myself (again, because I think I used parts we didn't cover in class yet) I add my example pseudo code for the methods of *Airplane* in Figure 5 below.

```
# Domain-related Operation: Registering an Airplane Maintenance

FUNCTION registerMaintenance()
       CALL updateLastMaintenance()
       CALL updateNextMaintenance()
END FUNCTION

FUNCTION updateLastMaintenance()
       SET lastMaintenanceDate TO currentDate
       PRINT "Last maintenance of airplane " + airplaneID + " has been set to " +
       lastMaintenanceDate + "."
END FUNCTION

FUNCTION updateNextMaintenance()
       SET nextMaintenanceDate TO lastMaintenanceDate + maintenanceInterval
       PRINT "The next maintenance of airplane " + airplaneID + " has been set to " +
       nextMaintenanceDate + "."
END FUNCTION
```

Figure 5: Pseudo code for the methods of class Airplane.

Composition relationship

Here I have chosen the relationship between *Book* and *Chapter*. It is a composition, because if a *Book* is deleted, so are its *Chapter* instances. It is a One-To-Many with minimum 1 relationship, because our stakeholders don't have instances of *Book* without any *Chapter.*

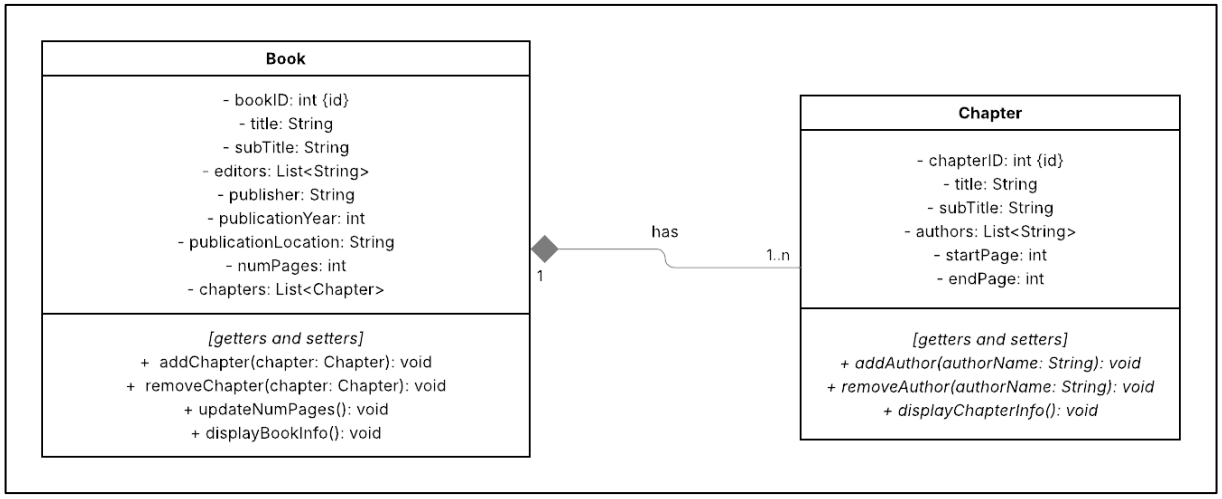

Figure 6: UML class-diagram for the chosen composition relationship.

It would be better to also have *Author* as a class, but for this exercise it should do as it is now. No further explanation here, because it would be redundant.